- 大数据知名培训专家、Spark大数据畅销书《**大数据Spark企业级实战**》作者王家林新作。
- 内容广度和深度兼顾，覆盖了**Spark**技术的核心知识点。
- 全程注重从架构的底层到上层，由宏观到微观的讲解。
- 秉承“**实战**”类图书特色，解析大量**案例和代码的编写操作。**
- 具有较强的可操作性，便于读者学习和理解。

Spark大数据实例开发教程

王家林　徐香玉　等编著



机械工业出版社
CHINA MACHINE PRESS

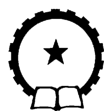


VISIT…

LANZAROTE Caliente.COM

Spark 大数据实例开发教程

王家林　徐香玉　等编著



机　械　工　业　出　版　社

本书是面向 Spark 开发者的一本实用参考书，书中结合实例系统地介绍了 Spark 的开发与使用。

本书包括 5 章内容，第 1 章为 Spark 简介；第 2 章为 Spark RDD 实践案例与解析；第 3 章为 Spark SQL 实践案例与解析；第 4 章为 Spark Streaming 实践案例与解析；第 5 章为 Tachyon 实战案例与解析。在全书最后的附录部分介绍了 Spark1. 4 版本的新特性。

本书适合刚接触 Spark 或对 Spark 分布式计算的开发不熟悉的初学者学习。对于熟悉函数式开发或面向对象开发，并有一定经验的开发者，本书也可以作为参考书。

图书在版编目(CIP)数据

Spark 大数据实例开发教程/王家林等编著. —北京：机械工业出版社，2015. 10

ISBN 978 – 7 – 111 – 51909 – 6

Ⅰ. ①S… Ⅱ. ①王… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字(2015)第 252644 号

机械工业出版社(北京市百万庄大街 22 号 邮政编码 100037)
策划编辑：王 斌 责任编辑：张 恒
责任校对：张艳霞 责任印制：李 洋
保定市中画美凯印刷有限公司印刷

2015 年 11 月第 1 版 · 第 1 次印刷
184mm × 260mm · 21. 25 印张 · 524 千字
0001– 4000册
标准书号：ISBN 978 – 7 – 111 – 51909 – 6
定价：59.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
电话服务
服务咨询热线：(010)88361066
读者购书热线：(010)68326294
(010)88379203
网络服务
机 工 官 网：www. cmpbook. com
机 工 官 博：weibo. com/cmp1952
教育服务网：www. cmpedu. com
金 书 网：www. golden – book. com
封面无防伪标均为盗版

前 言



Spark 起源于 2009 年，是美国加州大学伯克利分校 AMP 实验室的一个研究性项目。Spark 于 2010 年开源，是当今大数据领域最活跃、最热门、最高效的大数据通用计算平台，是 Apache 软件基金会所有开源项目中三大顶级开源项目之一。

Spark 是用 Scala 语言写成的一套分布式内存迭代计算系统，它的核心抽象概念是弹性分布式数据集（Resilient Distributed Dataset，RDD），在“One Stack to rule them all”（一个技术堆栈容纳各种数据处理技术）理念的指引下，Spark 基于 RDD 成功地构建起了大数据处理的一体化解决方案，将 MapReduce、Streaming、SQL、Machine Learning、Graph Processing 等大数据计算模型统一到一个技术堆栈中，开发者可以使用同样的 API 操作 Spark 中的所有功能。更为重要的是，Spark 的 Spark SQL、MLLib、GraphX、Spark Streaming 等四大子框架（在 Spark 1.4 版本中，加入了新的 SparkR 子框架）之间可以在内存中完美的无缝集成并可以互相操作彼此的数据，这不仅打造了 Spark 在当今大数据计算领域相比其他任何计算框架具备的无可匹敌的优势，更使得 Spark 正在加速成为大数据处理中心首选的和唯一的计算平台。

目前，Spark 已经发展成为包含众多子项目的大数据计算平台。Spark 的整个生态系统称为伯克利数据分析栈（BDAS）。其核心框架是 Spark，同时 BDAS 涵盖支持结构化数据 SQL 查询与分析的查询引擎 Spark SQL，提供具有机器学习功能的系统 MLbase 及底层的分布式机器学习库 MLlib、并行图计算框架 GraphX、流计算框架 Spark Streaming、采样近似计算查询引擎 BlinkDB、内存分布式文件系统 Tachyon、资源管理框架 Mesos 等子项目。这些子项目在 Spark 上提供了更高层、更丰富的计算范式。

随着 Spark 社区的不断成熟，它已被广泛应用于阿里巴巴、百度、网易、英特尔等各大公司的生产环境中。

关于 Spark 及其开发案例的中文资料比较匮乏，相关书籍也比较少，社区内开发者们主要的学习方式仍然限于阅读有限的官方文档、源码、AMPLab 发表的论文，以及社区讨论等。

为了让 Spark 初学者能快速进入开发阶段，本书针对 Spark 内核、Spark SQL 以及 Spark Streaming 等内容，提供了一系列的开发案例，基于这些开发案例，详细记录并解析了这几个子框架开发过程的各个步骤。

Spark 的发展日新月异，在本书撰写时，Spark 1.3 版本刚刚发布，因此，本书全部的开发案例都是基于该版本进行的。同时，鉴于 Spark 是用 Scala 语言编写的，本书的开发案例也采用 Scala 语言作为开发语言。

本书共 5 章，内容包括：

第 1 章：Spark 简介，内容包括介绍 Spark 的基本概念、Spark 生态圈以及 RDD 编程模型等内容；

第 2 章：Spark RDD 实践案例与解析，内容包括 Spark 应用程序的部署、RDD 数据的输

入、处理、输出的基本案例与解析、RDD API 的应用案例与解析、Spark 应用程序的构建，以及移动互联网数据分析案例与解析等内容；

第 3 章：Spark SQL 实践案例与解析，内容包括 Spark SQL 概述、DataFrame 处理的案例与解析、Spark SQL 处理各种数据源的案例与解析，以及基于 Hive 的人力资源系统数据处理案例与解析等内容；

第 4 章：Spark Streaming 实践案例与解析，内容包括 Spark Streaming 概述、Spark Streaming 基础概念、企业信息实时处理的案例与解析，以及性能调优等内容；

第 5 章：Tachyon 实践案例与解析，内容包括 Tachyon 概述、Tachyon 部署的案例与解析、Tachyon 配置的案例与解析、命令行接口的案例与解析、同步底层文件系统的案例与解析，以及基于 Tachyon 运行 Spark 和 Hadoop 的案例与解析等内容。

在全书最后，特别介绍了 Spark 1.4 版本的新特性。

预备知识

熟悉 Linux/UNIX 类操作系统的基本命令操作以及 Java 或 Scala 语言对理解本书内容大有裨益。建议构建 3 台及以上服务器的集群环境，以更好地实践并理解分布式环境中的 Spark 运行框架与计算。

本书的目标读者

作为 Spark 入门的开发案例，本书适合刚接触 Spark 或对 Spark 分布式计算的开发不熟悉的初学者。对于熟悉函数式开发或面向对象开发，并有一定经验的开发者，本书也可以作为开发案例的参考书籍。

本书由王家林，徐香玉编著，参与编写的还有：王家虎、王家俊、王燕军。限于作者水平，书中疏漏之处在所难免，欢迎广大读者批评指正。

编　者

目　　录



前言

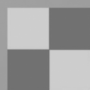
第 1 章　Spark 简介 ………………………………………………………… 1
1.1　什么是 Spark ………………………………………………………… 2
1.2　Spark 生态圈 ………………………………………………………… 2
1.2.1　伯克利数据分析协议栈 ………………………………………… 2
1.2.2　Spark 开源社区发展 …………………………………………… 3
1.3　RDD 编程模型 ……………………………………………………… 3
1.3.1　RDD 抽象概念 …………………………………………………… 3
1.3.2　RDD 的操作 ……………………………………………………… 5
1.3.3　RDD 的依赖关系 ………………………………………………… 6
1.3.4　一个典型的 DAG 示意图 ………………………………………… 6
第 2 章　Spark RDD 实践案例与解析 ……………………………………… 8
2.1　Spark 应用程序部署 ………………………………………………… 9
2.1.1　Spark 应用的基本概念 …………………………………………… 9
2.1.2　应用程序的部署方式 …………………………………………… 10
2.2　RDD 数据的输入、处理、输出的基本案例与解析 ………………… 14
2.2.1　集群环境的搭建 ………………………………………………… 15
2.2.2　交互式工具的启动 ……………………………………………… 19
2.2.3　文本数据的 ETL 案例实践与解析 ……………………………… 25
2.2.4　文本数据的初步统计案例实践与解析 ………………………… 28
2.2.5　文本数据统计结果的持久化案例实践与解析 ………………… 31
2.2.6　RDD 的 Lineage 关系的案例与源码解析 ……………………… 33
2.2.7　RDD 的持久化案例与解析 ……………………………………… 43
2.2.8　RDD 的构建案例与解析 ………………………………………… 48
2.2.9　分区数设置的案例与源码解析 ………………………………… 49
2.3　RDD API 的应用案例与解析 ………………………………………… 53
2.3.1　如何查找 RDD API 的隐式转换 ………………………………… 54
2.3.2　RDD[T]的分区相关的 API ……………………………………… 57
2.3.3　RDD[T]常用的聚合 API ………………………………………… 60
2.3.4　DoubleRDDFunctions(self:RDD[Double])常用的 API ……… 63
2.3.5　PairRDDFunctions[K,V]聚合相关的 API ……………………… 66
2.3.6　RDD 相互间操作的 API ………………………………………… 71
2.3.7　PairRDDFunctions[K,V]间的相关 API ………………………… 76
2.3.8　OrderedRDDFunctions[K,V,P <:Product2[K,V]]常用的 API …… 77

2.4　Spark 应用程序构建 …… 78
2.4.1　基于 SBT 构建 Spark 应用程序的实例 …… 79
2.4.2　基于 IDEA 构建 Spark 应用程序的实例 …… 81
2.4.3　Spark 提交应用的调试实例 …… 93
2.5　移动互联网数据分析案例与解析 …… 98
2.5.1　移动互联网数据的准备 …… 99
2.5.2　移动互联网数据分析与解析 …… 100
2.6　Spark RDD 实践中的常见问题与解答 …… 103
第 3 章　Spark SQL 实践案例与解析 …… 105
3.1　Spark SQL 概述 …… 106
3.2　DataFrame 处理的案例与解析 …… 106
3.2.1　DataFrame 编程模型 …… 107
3.2.2　DataFrame 基本操作案例与解析 …… 107
3.2.3　DataFrame 与 RDD 之间的转换案例与解析 …… 122
3.2.4　缓存表（列式存储）的案例与解析 …… 127
3.2.5　DataFrame API 的应用案例与分析 …… 132
3.3　Spark SQL 处理各种数据源的案例与解析 …… 158
3.3.1　通用的加载/保存功能的案例与解析 …… 160
3.3.2　Parquet 文件处理的案例与解析 …… 165
3.3.3　JSON 数据集操作的案例与解析 …… 167
3.3.4　操作 Hive 表的案例与解析 …… 170
3.3.5　使用 JDBC 操作其他数据库的案例与解析 …… 185
3.3.6　集成 Hive 数据仓库的案例与解析 …… 191
3.4　基于 Hive 的人力资源系统数据处理案例与解析 …… 197
3.4.1　人力资源系统的数据库与表的构建 …… 199
3.4.2　人力资源系统的数据的加载 …… 201
3.4.3　人力资源系统的数据的查询 …… 202
第 4 章　Spark Streaming 实践案例与解析 …… 206
4.1　Spark Streaming 概述 …… 207
4.2　Spark Streaming 基础概念 …… 208
4.3　企业信息实时处理的案例与解析 …… 208
4.3.1　处理 TCP 数据源的案例与解析 …… 209
4.3.2　处理 HDFS 文件数据源的案例与解析 …… 225
4.3.3　处理 Kafka 数据源的准备工作 …… 229
4.3.4　基于 Receiver 读取 Kafka 数据的案例与解析 …… 232
4.3.5　直接读取（无 Receiver）Kafka 数据的案例与解析 …… 243
4.3.6　处理 Flume 数据源的实践准备 …… 253
4.3.7　基于 Flume 风格的推送数据案例与解析 …… 254
4.3.8　定制 FlumeSink 的拉取数据案例与解析 …… 261

4.4　性能调优 …… 271
4.4.1　减少批处理的时间 …… 271
4.4.2　设置正确的批间隔 …… 273
4.4.3　内存调优 …… 274
第 5 章　Tachyon 实践案例与解析 …… 276
5.1　Tachyon 概述 …… 277
5.2　重新编译部署包 …… 279
5.2.1　重新编译 Tachyon 的部署包 …… 279
5.2.2　重新编译 Spark 的部署包 …… 279
5.3　Tachyon 部署的案例与解析 …… 283
5.3.1　单机模式部署的案例与解析 …… 283
5.3.2　集群模式部署的案例与解析 …… 291
5.3.3　集群 Master 容错部署的案例与解析 …… 294
5.4　Tachyon 配置的案例与解析 …… 299
5.4.1　底层存储系统的配置案例与解析 …… 299
5.4.2　配置属性与解析 …… 302
5.5　命令行接口的案例与解析 …… 306
5.5.1　命令行接口的说明 …… 306
5.5.2　命令行接口的案例实践与解析 …… 308
5.6　同步底层文件系统的案例与解析 …… 312
5.6.1　同步 HDFS 底层文件系统的案例与解析 …… 313
5.6.2　同步本地底层文件系统的案例与解析 …… 314
5.7　基于 Tachyon 运行的案例与解析 …… 316
5.7.1　基于 Tachyon 运行 Spark 的案例与解析 …… 316
5.7.2　基于 Tachyon 运行 Hadoop MR 的案例与解析 …… 327
附录　Spark 1.4 版本新特性 …… 330

第1章 Spark简介

1.1 什么是 Spark
1.2 Spark 生态圈
1.3 RDD 编程模型



Section 1.1 什么是 Spark

Spark 是一个由加州大学伯克利分校（UC Berkeley AMP）开发的一站式通用大数据计算框架，经过 2013～2014 年的高速发展，Spark 目前已经成为大数据计算领域最热门的技术之一。Spark 的核心技术弹性分布式数据集（Resilient Distributed Datasets，RDD），提供了比 Hadoop 更加丰富的 MapReduce 模型，拥有 HadoopMapReduce 所具有的所有优点，但不同于 HadoopMapReduce 的是，Spark 中 Job 的中间输出和结果可以保存在内存中，从而可以基于内存快速的对数据集进行多次迭代，来支持复杂的机器学习、图计算和准实时流处理等，效率更高，速度更快。

Section 1.2 Spark 生态圈

Spark 基于 RDD，提供了一站式多维度的大数据计算模型，可以在同一个技术堆栈中快速对数据集进行批处理、即席查询、机器学习、图计算和准实时流处理等。

1.2.1 伯克利数据分析协议栈

目前，Spark 已经发展成为包含众多子项目的大数据计算平台。Spark 的整个生态系统称为伯克利数据分析栈（BDAS）。其核心框架是 Spark，同时 BDAS 涵盖支持结构化数据 SQL 查询与分析的查询引擎 Spark SQL，提供具有机器学习功能的系统 MLbase 及底层的分布式机器学习库 MLlib、并行图计算框架 GraphX、流计算框架 Spark Streaming、采样近似计算查询引擎 BlinkDB、内存分布式文件系统 Tachyon、资源管理框架 Mesos 等子项目。这些子项目在 Spark 上层提供了更高层、更丰富的计算范式。BDAS 如图 1.1 所示。

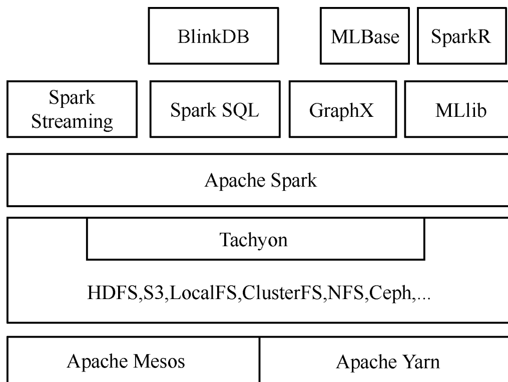


图 1.1 伯克利数据分析协议栈

1.2.2 Spark 开源社区发展

图 1.2 显示了自从 Spark 将其代码部署到 GitHub 之后的提交数据，其中包含了当前提交个数、分支个数、发布的 release 版本个数以及当前贡献者的个数。截止 2015 年 3 月份一共有 10150 次提交，13 个分支，33 次发布，469 位代码贡献者。这些数据可以看出 Spark 开源社区的活跃程度相当高。



图 1.2　Spark 开源提交数据

图 1.3 为截止到 2015 年 3 月份的 Spark 代码贡献者每个月的增长曲线，由该图可以看到 Spark 开源项目的代码贡献者在近几年增长很快，尤其是最近两年，越来越多的人参与在 Spark 开源项目中。

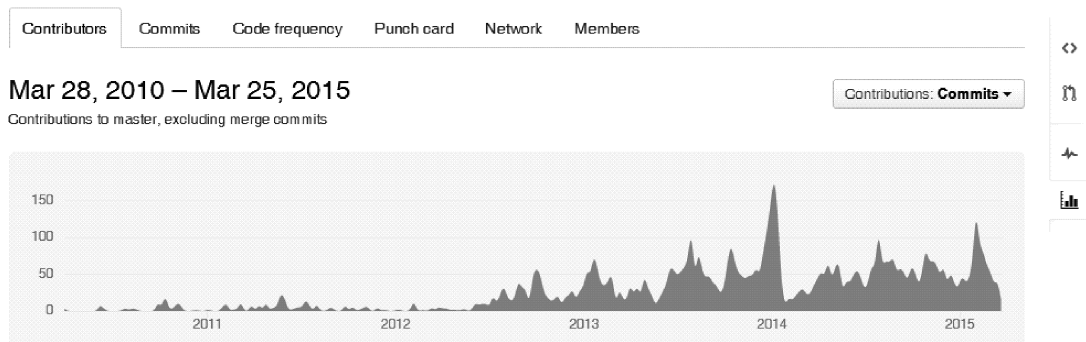


图 1.3　Spark 代码贡献者的增长曲线

Section 1.3 RDD 编程模型

Spark 的理论基础是 RDD，RDD 让 Spark 实现了“one stack to rule them all”（一个技术堆栈统一数据处理）目标。只有深刻理解 RDD 这一抽象概念，才能进一步理解 Spark 的有向无环图（Directed Acydic Graph，DAG）调度及 RDD 间的 Lineage（血统）关系，下面开始解析 RDD 抽象概念以及 DAG 调度、Lineage 机制等内容。

1.3.1 RDD 抽象概念

RDD 是弹性分布式数据集的简称，其本身是一个抽象类，其内部实现包括以下五个部分，其中前三个是必备的：

1）getPartitions 方法：分区列表（数据块列表）。

2）compute 方法：计算每个分片的函数。

3）getDependencies 方法：对父 RDD 的依赖列表。

4）partitioner：Key – Value（键 – 值）RDD 的分区器。

5）getPreferredLocations 方法：每个数据分片的预定义地址列表（如 HDFS 上的数据块的地址）。

其中，前三个用于描述 RDD 间的 Lineage（即血统关系，此概念在后面有专门解释）信息，后两个可用于优化执行。

RDD 是 Spark 中最重要的一个抽象概念，为了更容易理解 RDD 这一抽象概念，这里给出比较通俗的描述。

1. 首先 RDD 的定义为 RDD［T］，可以将 RDD 理解成 T 实例的一个集合，即 RDD 中的每条记录都是一个 T 实例，比如 T 为 String 时，RDD 就是一组 String 字符串的集合。

2. 对应的分区就是将这一组 T 实例的集合拆分成多个子集合，这里子集合也就是我们的数据分区。数据以 Block，即块方式存储在 HDFS 上，加载后，在 Spark 中，子集合实际上对应着分区的概念。分区就是将对应大数据量的 T 实例集合切（split）成多个小数据量的 T 实例子集合。这个集合，对应的内部代码其实就是 Iterator［T］。

3. 用于构建该 RDD 的父 RDD 即是该 RDD 的父依赖，由于可以有多个父依赖的 RDD，因此有对应父 RDD 的一个依赖列表。对于 RDD 间的依赖关系，首先需要理解一个概念，就是依赖这个概念是通过 RDD 的分区间的依赖来体现的，通过这个依赖列表，以及该 RDD 的 getPartitions 方法，可以知道 RDD 的各个分区是如何依赖一组父 RDD 的分区的，比如最简单的映射（map）转换，转换后的 RDD（MapPartitionsRDD）各个分区，依赖父 RDD 的各个分区（分区为一一对应的依赖关系）。也就是转换后 RDD 各个分区的数据 Block，是来自父 RDD 的对应分区的数据 Block。

4. 计算每个分片的函数 compute，体现了惰性（lazy）计算的特性，比如：MapPartitionsRDD，对应的 compute 函数记录了该 RDD 对父依赖的各个分区数据的操作，也就是记录了对 MapPartitionsRDD 各个分区的输入源数据进行的计算。当其他 RDD 从 MapPartitionsRDD 获取数据或要存储到外部存储系统时（有 Action 触发），就会执行这个计算每个分片的函数 compute。因为这里只是记录操作，所以在窄依赖时可以采用 pipeline 方式，流水线式地进行计算。

这里再补充介绍两点：

1）compute 函数是针对分区的数据，所以计算的并行数也就是分区的个数。

2）compute 函数是针对分区的数据，可以认为计算的粒度是分区粒度，因此可以认为 RDD 的计算，某种程度上是粗粒度的。也就是如果使用 RDD 的 compute 函数，则大部分 API 的计算都是针对分区的，而不是针对 RDD 的 T 元素。如果需要对 RDD 的 T 元素进行更细粒度的处理，可以使用 RDD 的 mapPartitions 操作，直接处理分区的数据 Iterator[T]。

5. Key – Value RDD，其实就是 T 类型为 Key – Value 对的类型。属于 RDD 元素是键值对这一特定类型的 RDD。RDD 为一些特定类型的 T 提供了额外的功能，这部分内容可以参考本书 2.3 节 RDD API 的应用案例与解析部分。

RDD 抽象类的这五个方法对应的源码如下所示：

```
/**
```

```
  * ::DeveloperApi::
  * Implemented by subclasses to compute a given partition.
  */
@DeveloperApi
def compute(split:Partition,context:TaskContext):Iterator[T]

/**
  * Implemented by subclasses to return the set of partitions in this RDD. This method will only
  * be called once,so it is safe to implement a time - consuming computation in it.
  */
protected def getPartitions:Array[Partition]

/**
  * Implemented by subclasses to return how this RDD depends on parent RDDs. This method will only
  * be called once,so it is safe to implement a time - consuming computation in it.
  */
protected def getDependencies:Seq[Dependency[_]] = deps

/**
  * Optionally overridden by subclasses to specify placement preferences.
  */
protected def getPreferredLocations(split:Partition):Seq[String] = Nil

/** Optionally overridden by subclasses to specify how they are partitioned. */
@transient val partitioner:Option[Partitioner] = None
```

1.3.2 RDD 的操作

RDD 的操作分为两类：Transformation 与 Action。其中 Transformations 是惰性执行（lazy execution）的，惰性执行表示真正需要时才会执行，这里是在需要具体的 Action 去触发才会开始执行，每个 Action 的触发都会提交一个 Job。

一个典型的操作流程如图 1.4 所示。

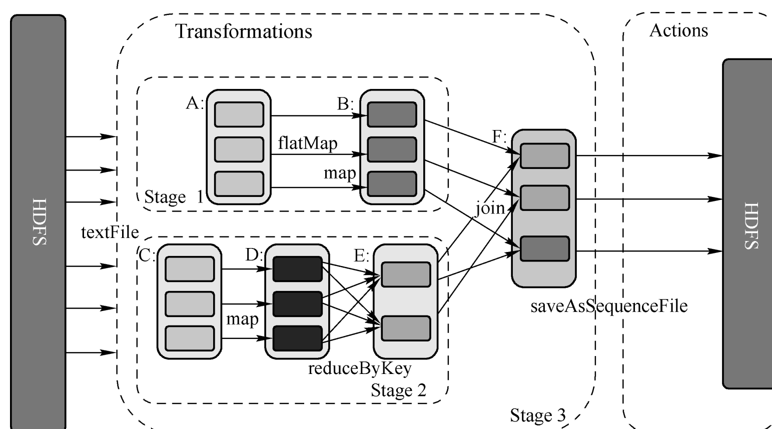


图 1.4 RDD 操作流程示意图

首先通过 textFile 操作从外部存储系统 HDFS 中读取文件，构建出两个 RDD 实例 A 和 C；然后 A 做 flatMap 和 Map 转换操作，对 C 做 Map 型操作和 reduceByKey 转换操作；最后对得到的 B 和 E 两个做联合操作，并通过 saveAsSequenceFile 操作将最终的 F 实例持久化到外部存储系统 HDFS 上。Transformation 与 Action 两种操作的区分可以从它们的返回值查看，Transformation 是将一个 RDD 转换为新的 RDD，而 Action 操作会将结果反馈到 Driver Program 或存储到外部存储系统上。

1.3.3 RDD 的依赖关系

RDD 的依赖分为窄依赖与宽依赖，如图 1.5 所示（左侧为窄依赖，右侧为宽依赖）：

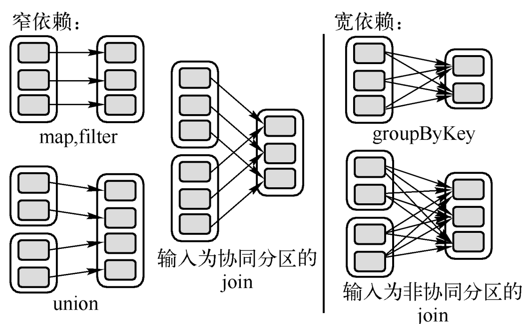


图 1.5　RDD 的窄依赖与宽依赖

其中，每一个方框表示一个 RDD，其内的阴影矩形表示 RDD 的分区。

对于窄依赖，可以进行 pipeline 操作，即允许在单个集群节点上流水线式地执行，这个节点可以计算所有父级分区。

而在节点失败（如节点硬件故障或强制 kill 导致的失败）后的恢复效率上，在窄依赖中，只有在失败节点上丢失的父级分区需要重新计算，并且这些丢失的父级分区可以并行地在不同节点上重新计算。与此相反，在宽依赖的继承关系中，单个失败的节点可能导致一个 RDD 的所有先祖 RDD 中的一些分区丢失，导致计算的重新执行。

对 RDD 依赖可以从以下两个方面理解：

1. 依赖本身是描述两个 RDD 之间的关系。但一个 RDD 可以与多个 RDD 有依赖关系。
2. 宽依赖和窄依赖的判断：在 RDD 的各个分区中对父 RDD 的分区的依赖关系。

1）窄依赖：子 RDD 的每个分区依赖于常数个父分区（即与数据规模无关）。

2）宽依赖：子 RDD 的每个分区依赖于所有父 RDD 的分区。

1.3.4 一个典型的 DAG 示意图

Spark 将数据在分布式环境下分区，然后将作业转化为 DAG，并分阶段进行 DAG 的调度和任务的分布式并行处理。图 1.6 是一个典型的 DAG 示意图。

如图 1.6 所示，描述了 DAG 调度时会根据 Shuffle 将 Job 划分为 Stage，比如 A 到 B 之间，以及 F 到 G 之间的数据需要经过 Shuffle 过程，因此 A 和 F 是 Stage 的划分点，以及

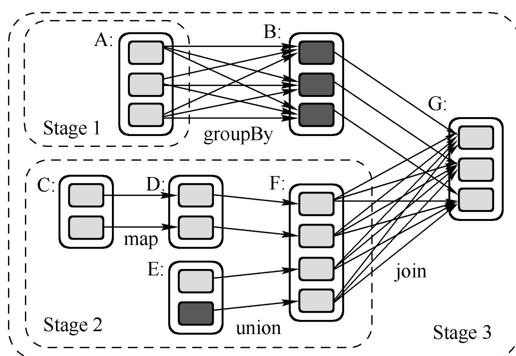


图 1.6　DAG 典型示意图

RDD 的 Lineage 关系。其中，实线圆角方框标识的是 RDD，方框中的矩形块为分区。

这里通过 G 和 F 这两个 RDD 间的依赖关系，描述了如何执行 Stage。图 1.6 中，RDD G 对 F 的依赖为宽依赖，即对应有 Shuffle 过程，因此对于 G 这个 RDD 就会新建一个 Stage，这里为 Stage 3。

RDD 的 Lineage 关系，可以从族谱角度去理解。即描述了 RDD 是从哪来的，以及怎么来的信息。

RDD G 上一个 Action 的执行将会以宽依赖作为基于为分区来构建各个 Stage，对各 Stage 内部的窄依赖则前后连接构成流水线。在本例中，Stage 1 的输出已经存在 RAM 中，所以直接执行 Stage 2 ，然后 Stage 3。

Spark 通过 Lineage 机制实现高容错，基于 DAG 图，Lineage 是轻量级而高效的，操作之间相互具备 Lineage 的关系，每个操作只关心其父操作，各个分片的数据之间互不影响，出现错误的时候只要恢复单个分片或分区即可。

DAG 的 Lineage 机制可以从两个方面进行理解：

1）RDD 之间的数据流图，即 RDD 的各个分区的数据是从哪来的。

2）基于数据流图之上的操作算子流图，即这些数据传递过来的时候经过了哪些算子的操作。

可以从 RDD 的抽象概念来解析 Lineage 的这两个方面的特性：

1）RDD 的分区列表（数据块列表）和对父 RDD 的依赖列表：对应 RDD 类的 getPartitions 和 getDependencies，这两个方法记录了该 RDD 的数据来源，以及来源数据如何获取（对应窄依赖和宽依赖）。

其中，RDD 各个分区的数据来源可以从外部存储系统或 Scala 数据集获取，也可以从其他父 RDD 获取。

2）计算每个分片的函数：对应 RDD 类的 compute 方法，该方法记录了该 RDD 对它各个分区的数据来源进行的计算。

第 2 章　Spark RDD实践案例与解析

2.1　Spark 应用程序部署

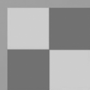
2.2　RDD 数据的输入、处理、输出的基本案例与解析

2.3　RDD API 的应用案例与解析

2.4　Spark 应用程序构建

2.5　移动互联网数据分析案例与解析

2.6　Spark RDD 实践中的常问问题与解答



Section
2.1 Spark 应用程序部署

图 2.1 描述了 Spark 应用程序在集群上的部署架构。

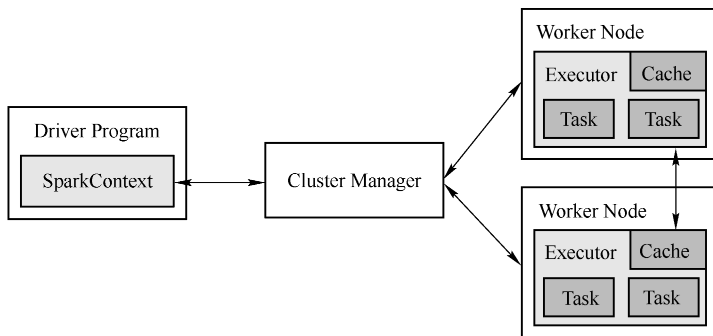


图 2.1　Spark 应用程序的部署架构图

在图 2.1 中，应用程序在集群上的部署架构包含以下内容：

1. Spark 应用程序，对应图中的 Driver Program，由于 SparkContext 包含在 Driver Program 中，因此通常也用 SparkContext 表示 Spark 应用程序。

2. 集群管理器，对应图中的 Cluster Manager，负责封装不同的集群管理器，包括 Spark Standalone 集群管理器、YARN 集群管理器等；Driver Program 通过 Cluster Manager 为其分配资源，然后将任务发送到多个 Worker Node 上执行。

3. 节点集群中多个 Worker Node，这是应用程序运行时真正执行应用代码的地方。这是分布式环境，应用程序在运行时的 Task 是在 Worker Node 上的 Executor 中执行。

部署的应用程序在逻辑上由 Driver Program 和运行在多个 Worker Node 上的 Executor 组成。

需要注意的是 Spark Driver 所在的计算机需要和 Spark 集群位于同一个网络环境中，因为 Driver 中的 SparkContext 实例要发送任务给不同 Worker Node 的 Executor 并接受 Executor 的一些执行结果信息，一般而言，在企业实际的生产环境中 Driver 所在计算机的性能配置往往都是比较不错的，尤其是其 CPU 的处理能力往往都很强悍。

2.1.1 Spark 应用的基本概念

Spark 应用程序部署需要了解的概念参见表 2.1。

表 2.1　Spark 应用程序基本概念

概　　念	含　　义
RDD	Spark 的基本计算单元，是 Spark 的一个最核心的抽象概念，可以通过一系列算子进行操作，包括 Transformation 和 Action 两种算子操作
Application	即 Spark 应用程序，是指创建了 SparkContext 实例对象的 Spark 用户程序，包含了一个 Driver Program 和集群中多个 Worker Node 上的 Executor，其中，每个 Worker Node 为每个应用仅仅提供一个 Executor

续表

概　念	含　义
Driver Program	是指运行 Application 的 main 函数并且新建 SparkContext 实例的程序。通常用 SparkContext 代表 Driver Program
Executor	是 Worker Node 为 Application 启动的一个工作进程，在进程中负责任务（Task）的运行，并且负责将数据存放在内存或磁盘上，必须注意的是，每个应用在一个 Worker Node 上只会有一个 Executor，在 Executor 内部通过多线程的方式并发处理应用的任务
Job	和 Spark 的 Action 相对应，每一个 Action 例如 count、savaAsTextFile 等都会对应一个 Job 实例，该 Job 实例包含多任务的并行计算
Stage	一个 Job 会被拆分成多组任务（TaskSet），每一组任务被称为 Stage，任务和 MapReduce 的 Map 和 Reduce 任务很像 划分 Stage 的依据在于：Stage 开始一般是由于读取外部数据或者 Shuffle 数据，一个 Stage 的结束一般是由于发生 Shuffle（例如 reduceByKey 操作）或者是在整个 Job 结束时。例如要把数据放到 HDFS 等存储系统上
Task	被 Driver Program 送到 Executor 上的工作单元，通常情况下一个 Task 会处理一个 Split（也就是一个分区）的数据，每个 Split 一般就是一个 Block 块的大小

2.1.2 应用程序的部署方式

部署 Spark 应用程序的两种方式分别为：交互式的方式和提交应用的方式。

在不同的应用场景下，可以选择不同的部署方式。一般来说，在开发调试的场景下，往往选择交互式方式来部署应用；而在需要向集群提交应用的场景时，就应该选择提交应用方式了。

Spark 为这两种部署方式提供了对应的工具——交互式工具 spark - shell 和应用程序部署工具 spark - submit。其中，spark - shell 是一个应用程序，因为 spark - shell 在启动的时候创建了 SparkContext 对象，其名称为 sc。这两种部署方式基本上可以运行在各种集群部署上。

使用这两种方式部署应用程序时，需要根据不同需求设置不同的使用方法及其命令选项参数，下面分别描述两种方式下的命令使用方法以及命令选项设置。

一、交互方式部署 Spark 应用程序

在单机模式下使用 Spark Shell 交互工具部署 Spark 应用程序，是初学者开始代码实践的最简单的、最快捷的方法。在集群模式下，也可以通过 Spark Shell 交互工具让开发者更方便地调试代码。

在 Spark 集群部署主目录下，输入：

```
./bin/spark - shell -- help
```

可得：

```
"Usage:.\bin\spark - shell.cmd [options]"
Options:
   -- master MASTER_URL Spark://host:port,mesos://host:port,yarn,or local.
   -- deploy - mode DEPLOY_MODE Whether to launch the driver program locally ("client") or one of
the worker machines inside the cluster ("cluster") (Default:client).
   -- class CLASS_NAME                Your application' s main class (for Java / Scala apps).
```

```
--name NAME                    A name of your application.
--jars JARS                    Comma-separated list of local jars to include on the driver and executorclasspaths.
--packages                     Comma-separated list of maven coordinates of jars to includeon the driver and executorclasspaths. Will search the localmaven repo,then maven central and any additional remoterepositories given by --repositories. The format for thecoordinates should be groupId:artifactId:version.
--repositories                 Comma-separated list of additional remote repositories tosearch for the maven coordinates given with --packages.
--py-files PY_FILES            Comma-separated list of .zip,.egg,or .py files to place
on the PYTHONPATH for Python apps.
--files FILES                  Comma-separated list of files to be placed in the workingdirectory of each executor.
--conf PROP=VALUE              ArbitrarySpark configuration property.
--properties-file FILE         Path to a file from which to load extra properties. If notspecified,this will look for conf/Spark-defaults.conf.
--driver-memory MEM            Memory for driver (e.g. 1000M,2G) (Default:512M).
--driver-java-options          Extra Java options to pass to the driver.
--driver-library-path          Extra library path entries to pass to the driver.
--driver-class-path            Extra class path entries to pass to the driver. Note thatjars added with --jars are automatically included in theclasspath.
--executor-memory MEM          Memory per executor (e.g. 1000M,2G) (Default:1G).
--proxy-user NAME              User to impersonate when submitting the application.
--help, -h                     Show this help message and exit
--verbose, -v                  Print additional debug output
--version,                     Print the version of currentSpark
Spark standalone with cluster deploy mode only:
--driver-cores NUM             Cores for driver (Default:1).
--supervise                    If given,restarts the driver on failure.
--kill SUBMISSION_ID           If given,kills the driver specified.
--status SUBMISSION_ID         If given,requests the status of the driver specified.
Spark standalone and Mesos only:
--total-executor-coresNUM      Total cores for all executors.
YARN-only:
--driver-cores NUM             Number of cores used by the driver,only in cluster mode (Default:1).
--executor-cores NUM           Number of cores per executor (Default:1).
--queue QUEUE_NAME             The YARN queue to submit to (Default:"default").
--num-executors NUM            Number of executors to launch (Default:2).
--archives ARCHIVES            Comma separated list of archives to be extracted into theworking directory of each executor.
```

如果查看交互式工具 spark-shell 的代码的话，可以看到其命令选项的帮助信息是通过调用 spark-submit --help 来获取的，因此其参数和 spark-submit 是一样的。

二、提交应用方式部署 Spark 应用程序

采用提交应用的方式部署 Spark 应用程序时，应用程序使用提交脚本进行部署：$Spark_HOME/bin/spark-submit。提交部署时，应根据具体的集群模式，配置相应的选项参数。可通过在 Spark 集群部署的主目录下，输入以下命令查询相应的配置参数：

```
./bin/spark - submit -- help
```

应用程序部署工具 spark - submit 的用法如下：

```
Usage:Spark - submit [options] <app jar | python file> [app arguments]
Usage:Spark - submit -- kill [submission ID] -- master [Spark://...]
Usage:Spark - submit -- status [submission ID] -- master [Spark://...]

Options:
  -- master MASTER_URL Spark://host:port,mesos://host:port,yarn,or local.
  -- deploy - mode DEPLOY_MODE Whether to launch the driver program locally ("client") or one of the worker machines inside the cluster ("cluster") (Default:client).
  -- class CLASS_NAME          Your application' s main class (for Java / Scala apps).
  -- name NAME                 A name of your application.
  -- jars JARS                 Comma - separated list of local jars to include on the driver and executorclasspaths.
  -- packages                  Comma - separated list of maven coordinates of jars to includeon the driver and executorclasspaths. Will search the localmaven repo,then maven central and any additional remoterepositories given by -- repositories. The format for thecoordinates should be groupId:artifactId:version.
  -- repositories              Comma - separated list of additional remote repositories tosearch for the maven coordinates given with -- packages.
  -- py - files PY_FILES       Comma - separated list of .zip,.egg,or .py files to place
on the PYTHONPATH for Python apps.
  -- files FILES               Comma - separated list of files to be placed in the workingdirectory of each executor.
  -- conf PROP = VALUE         ArbitrarySpark configuration property.
  -- properties - file FILE    Path to a file from which to load extra properties. If notspecified,this will look for conf/Spark - defaults.conf.
  -- driver - memory MEM       Memory for driver (e.g. 1000M,2G) (Default:512M).
  -- driver - java - options   Extra Java options to pass to the driver.
  -- driver - library - path   Extra library path entries to pass to the driver.
  -- driver - class - path     Extra class path entries to pass to the driver. Note thatjars added with -- jars are automatically included in theclasspath.
  -- executor - memory MEM     Memory per executor (e.g. 1000M,2G) (Default:1G).
  -- proxy - user NAME         User to impersonate when submitting the application.
  -- help, - h                 Show this help message and exit
  -- verbose, - v              Print additional debug output
  -- version,                  Print the version of currentSpark
Spark standalone with cluster deploy mode only:
  -- driver - cores NUM        Cores for driver (Default:1).
  -- supervise                 If given,restarts the driver on failure.
  -- kill SUBMISSION_ID        If given,kills the driver specified.
  -- status SUBMISSION_ID      If given,requests the status of the driver specified.
Spark standalone and Mesos only:
  -- total - executor - coresNUM   Total cores for all executors.
YARN - only:
  -- driver - cores NUM        Number of cores used by the driver,only in cluster mode (Default:1)
.
```

```
  --executor-cores NUM          Number of cores per executor (Default:1).
  --queue QUEUE_NAME           The YARN queue to submit to (Default:"default").
  --num-executors NUM           Number of executors to launch (Default:2).
  --archives ARCHIVES          Comma separated list of archives to be extracted into theworking di-
rectory of each executor.
```

命令行选项可以在提交时动态修改配置属性，如果配置属性是常用的话，可以放置在 conf/spark-defaults. conf 文件下，当属性对应 JVM 配置时，即各种 JAVA 的 OPTS 等，可以在 conf/spark-env. sh 脚本中添加到对应环境变量中。

比如，命令行选项中的比较常用的--jars 选项，是逗号分隔的本地 jar 包列表，支持自动将这些 jar 包发布并放置到执行环境的 CLASS PATH 下。可以在提交时动态指定该选项，也可以在 conf/spark-defaults. conf 中设置相关的 CLASS PATH 信息，避免每次提交时都要动态指定，而且可以避--jars 选项带来的网络 I/O 等开销。可以查看官方网站上的配置页，部分相关的配置参见表 2. 2。

表 2. 2　CLASS_PATH 相关配置项

配 置 属 性	含　义
spark. driver. extraClassPath	Driver 额外的 CLASS PATH 设置。比如，在部署模式（--deploy-mode）为 cluster 时，可以将比较常用的驱动类 jar 包等部署到集群各个节点上，然后将部署路径设置到该配置属性中，来避免--jars 选项的使用
spark. driver. extraJavaOptions	Driver 额外的 JAVA_OPTS 设置
spark. driver. extraLibraryPath	Driver 额外的第三方库路径设置
spark. executor. extraClassPath	Executor 额外的 CLASS PATH 设置。比如，可以将比较常用的驱动类 jar 包等部署到集群各个节点上，然后将部署路径设置到该配置属性中，来避免--jars 选项的使用
spark. executor. extraJavaOptions	Executor 额外的 JAVA_OPTS 设置
spark. executor. extraLibraryPath	Executor 额外的第三方库路径设置

应用提交案例：

```
$ Spark_HOME/bin/spark-submit-class test. HelloSparkMaster --master spark://192. 168. 70. 214:
7077 /toPath/testprojectide_2. 10-1. 0. jar
```

其中：

1）test. HelloSparkMaster 为要运行的应用程序类的名称，必须是全路径的。运行 Spark 提供的 example 示例代码时，可以直接使用类名，这是因为在 spark-shell2. sh 脚本文件中已经自动添加了 example 示例类的全路径前缀”org. apache. Spark. examples. ”。

2）spark://192. 168. 70. 214:7077:MasterURL，如果是提交到 Spark Standalone 集群的话，该地址信息必须和集群 Web Interface 界面显示的 MasterURL 一样。如图 2. 2 所示，该 Master 会用于 akka（一个用 Scala 编写的库），作为 Actor 的地址，而该地址与实际使用 IP 值的 Master 构成的地址不同，这会导致 akka 通信失败报错，因此提交时必须保证 MasterURL 的正确性。

Spark 是 M/S 结构的分布式计算框架，M 为 Master，S 为 Slave，其中 Master 对应于

Spark Standalone 集群的调度节点。Master URL 对应连接到 Master 时使用的 URL。图 2.2 为 Web Interface（http://wxx214:8080/）界面，在该界面查看正确的 MasterURL。

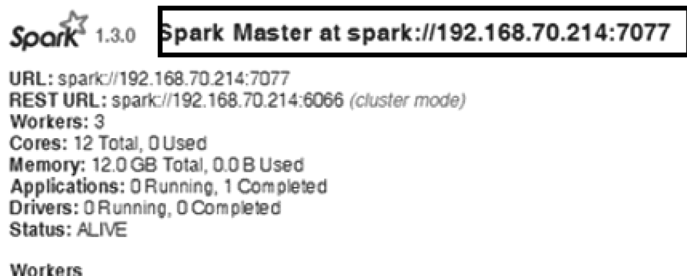


图 2.2　URL 查看界面

MasterURL 中的 host 信息不支持 IP 与 hostname 互换，即当界面显示如图 2.2 时，如果使用 hostname 对应的 MasterURL，比如以 Spark://wxx214:7077 作为 MasterURL 进行提交的话，可能会报如下连接失败的错误：

```
15/04/03 10:13:04 INFO ui. SparkUI:Started SparkUI at http://wxx215:4040
15/04/03 10:13:04 INFO client. AppClient $ ClientActor:Connecting to master akka. tcp://sparkMaster
@ wxx214:7077/user/Master...
15/04/03 10:13:24 INFO client. AppClient $ ClientActor:Connecting to master akka. tcp://sparkMaster
@ wxx214:7077/user/Master...
15/04/03 10:13:44 INFO client. AppClient $ ClientActor:Connecting to master akka. tcp://sparkMaster
@ wxx214:7077/user/Master...
15/04/03 10:14:04 ERROR cluster. SparkDeploySchedulerBackend:Application has been killed. Rea-
son:All masters are unresponsive! Giving up.
15/04/03 10:14:04 WARN cluster. SparkDeploySchedulerBackend:Application ID is not initialized yet.
15/04/03 10:14:04 ERROR scheduler. TaskSchedulerImpl:Exiting due to error from cluster scheduler:
All masters are unresponsive! Giving up.
```

3）/toPath/testprojectide_2.10 - 1.0.jar：应用程序类所在的 jar 包路径。

Section 2.2 RDD 数据的输入、处理、输出的基本案例与解析

本节给出 RDD 数据输入、处理以及输出的一个完整案例，案例中包含了以下内容：

1）构建 RDD：从外部存储系统的数据集构建。

2）对构建好的 RDD 进行处理：包括 WordCount（单词统计）、Sort（排序）、TopN（前 N 个）等常见应用处理。

3）保存 RDD：存储经过各种处理的 RDD 数据。

补充说明：当 Spark 以 Hadoop 的 HDFS 作为存储系统时，文件的读写依赖于 HDFS 提供的接口，文件的读取对应具体的 InputFormat 子类，文件的输出对应具体的 OutputFormat 子类。即，存储系统的操作由 HDFS 提供支持，Spark 本身只是计算框架。因此，当我们对输

入输出有特殊要求时，在构建 HadoopRDD 的时候，应使用对应的 InputFormat 子类和 OutputFormat 子类来实现。

2.2.1 集群环境的搭建

对初学者而言，只需要掌握以下几个启动、停止的命令，就可以相应地启动和停止对应的集群了。在接下来介绍的集群准备时，只启动 HDFS 和 Spark 两个服务。

一、集群规划

在四台机器上构建集群，具体环节搭建的规划如表 2.3 所示。

表 2.3 集群部署环境说明

IP 地址	Master/Slave	进 程	部署版本
192.168.70.214	Master	Hadoop：NameNode、ResourceManager Spark：Master	Hadoop –2.6.0、Spark –1.3.0 –bin –hadoop2.4
192.168.70.215	Slave	Hadoop：DataNode、NodeManager Spark：Worker	
192.168.70.223			
192.168.70.225			

二、版本说明

1. 支持环境采用 Hadoop –2.6.0 –64：64 位操作系统上重新编译的 Hadoop 2.6.0 版本的部署包。

2. Spark –1.3.0 –bin –hadoop2.4：基于 Hadoop 2.4 版本重新编译的 spark 1.3.0 版本的部署包。

由于 Hadoop 的客户端兼容二进制，因此基于 Hadoop 2.4 版本编译的 Spark 可以访问 Hadoop2.6 版本的 HDFS。

三、启动 Hadoop 支持环境

Hadoop 集群环境的搭建请参考王家林老师的《大数据 Spark 企业级实践》，在此不再赘述。

Hadoop 环境对 Spark 的支持主要有两方面：

1. 通过 HDFS 实现底层存储系统的支持

HDFS 是一个主/从（Mater/Slave）式的体系结构，从最终用户的角度来看，它就像传统的文件系统一样，可以通过目录路径对文件执行 CRUD（Create、Read、Update 和 Delete）操作。但由于分布式存储的性质，HDFS 集群拥有一个 NameNode 和一些 DataNode。NameNode 管理文件系统的元数据，DataNode 存储实际的数据。客户端通过同 NameNode 和 DataNodes 的交互访问文件系统。客户端通过访问 NameNode 来获取文件的元数据，而真正的文件 I/O 操作是直接和 DataNode 进行交互的。

2. 通过 Yarn 实现集群资源管理的支持

Yarn 是 Hadoop 2.0 新增的集群资源管理系统，负责集群的资源管理和调度，使得多种计算框架可以运行在一个集群中。

如果使用 Spark 自带的资源管理器，即使用 Spark Standalone 模式的话，只需要提供 Hadoop 的 HDFS 支持；如果同时使用了 Hadoop 的集群资源管理器，即采用 Spark on Yarn 模式的话，需要同时提供 Hadoop 的 Yarn。

在集群中的 Master 节点上，分别启动 HDFS 和 Yarn。

四、Hadoop HDFS 的格式化

Hadoop 的 HDFS 部署好了之后并不能马上使用，而是先要对配置的文件系统进行格式化。在这里要注意两个概念，一个是文件系统，此时的文件系统在物理上还不存在，或许用网络磁盘来描述会更加合适；二就是格式化，此处的格式化并不是指传统意义上的本地磁盘格式化，而是一些清除与准备工作。在 Hadoop 部署目录下输入命令：

```
[harli@ wxx214 hadoop -2. 6. 0] $ ./bin/hdfs - namenode - format
```

执行结果如下所示：

```
15/04/01 10:41:31 INFOnamenode. NameNode:STARTUP_MSG:
    /************************************************************
    STARTUP_MSG:   StartingNameNode
    STARTUP_MSG:   host = cluster01/192. 168. 242. 131
    STARTUP_MSG:   args = [ - format]
    STARTUP_MSG:   version = 2. 6. 0
    STARTUP_MSG:   classpath = /home/harli/cluster_13/hadoop/etc/hadoop:/home/harli/cluster_
13/hadoop/share/hadoop/common/lib/jackson - xc - 1. 9. 13. jar:……. .
    STARTUP_MSG:   build = Unknown - r Unknown;compiled by' root' on 2015 - 01 - 05T10:00Z
    STARTUP_MSG:   java = 1. 7. 0_71
```

五、启动集群的 HDFS 服务

格式化 NameNode 后，在 Master 节点用启动脚本启动集群 HDFS，在 Hadoop 部署目录下输入命令 ./sbin/start - dfs. sh，启动 dfs，运行该命令结果如下：

```
[harli@ wxx214 hadoop -2. 6. 0] $ ./sbin/start - dfs. sh
Startingnamenodes on [wxx214]
wxx214:starting namenode,logging to /home/harli/cluster_13/hadoop - 2. 6. 0/logs/hadoop - harli - namenode - wxx214. out
wxx223:starting datanode,logging to /home/harli/cluster_13/hadoop - 2. 6. 0/logs/hadoop - harli - datanode - wxx223. out
wxx225:starting datanode,logging to /home/harli/cluster_13/hadoop - 2. 6. 0/logs/hadoop - harli - datanode - wxx225. out
wxx215:starting datanode,logging to /home/harli/cluster_13/hadoop - 2. 6. 0/logs/hadoop - harli - datanode - wxx215. out
Starting secondarynamenodes [wxx214]
wxx214:starting secondarynamenode,logging to /home/harli/cluster_13/hadoop - 2. 6. 0/logs/hadoop - harli - secondarynamenode - wxx214. out
[harli@ wxx214 hadoop -2. 6. 0] $ jps
29524 Jps
28542ResourceManager
26646TachyonMaster
29404SecondaryNameNode
29212NameNode
```

```
9373 Master
```

对应的停止 dfs 的命令为./sbin/stop-dfs.sh，运行该命令结果如下：

```
[harli@wxx214 hadoop-2.6.0]$ ./sbin/stop-dfs.sh
Stoppingnamenodes on [wxx214]
wxx214:stopping namenode
wxx223:stopping datanode
wxx225:stopping datanode
wxx215:stopping datanode
Stopping secondarynamenodes [wxx214]
wxx214:stopping secondarynamenode
[harli@wxx214 hadoop-2.6.0]$ jps
30144 Jps
28542ResourceManager
26646TachyonMaster
9373 Master
```



六、启动集群的 Yarn 服务

如果要使用 Spark on Yarn 模式，在 Hadoop 的资源管理器 Yarn 下提交应用的话，可以通过 Yarn 启动脚本来启动集群的 Yarn，在 Hadoop 部署目录下输入命令./sbin/start-yarn.sh，运行该命令结果如下：

```
[harli@wxx214 hadoop-2.6.0]$ ./sbin/start-yarn.sh
starting yarn daemons
startingresourcemanager,logging to /home/harli/cluster_13/hadoop-2.6.0/logs/yarn-harli-resource-
manager-wxx214.out
wxx223:starting nodemanager,logging to /home/harli/cluster_13/hadoop-2.6.0/logs/yarn-harli-
nodemanager-wxx223.out
wxx225:starting nodemanager,logging to /home/harli/cluster_13/hadoop-2.6.0/logs/yarn-harli-
nodemanager-wxx225.out
wxx215:starting nodemanager,logging to /home/harli/cluster_13/hadoop-2.6.0/logs/yarn-harli-
nodemanager-wxx215.out
[harli@wxx214 hadoop-2.6.0]$ jps
26646TachyonMaster
27935 Jps
27669ResourceManager
9373 Master
```

对应的停止 Yarn 命令为./sbin/stop-yarn.sh，运行代码如下：

```
[harli@wxx214 hadoop-2.6.0]$ ./sbin/stop-yarn.sh
stopping yarn daemons
stoppingresourcemanager
wxx223:stopping nodemanager
wxx225:stopping nodemanager
wxx215:stopping nodemanager
noproxyserver to stop
```

从启动和停止后的 Jps 信息中，可以看出对应服务的进程信息。

七、启动 Spark 集群

Spark 集群环境的搭建请参考王家林老师的《大数据 Spark 企业级实践》一书。

进入 Spark 部署目录，输入命令 start - all，启动 Spark 集群：

```
[harli@ wxx214spark - 1. 3. 0 - bin - hadoop2. 4] $ ./sbin/start - all. sh
starting org. apache. spark. deploy. master. Master, logging to /home/harli/cluster_13/spark - 1. 3. 0 - bin - hadoop2. 4/sbin/. . /logs/spark - harli - org. apache. spark. deploy. master. Master - 1 - wxx214. out
wxx225: starting org. apache. spark. deploy. worker. Worker, logging to /home/harli/cluster_13/spark - 1. 3. 0 - bin - hadoop2. 4/sbin/. . /logs/spark - harli - org. apache. spark. deploy. worker. Worker - 1 - wxx225. out
wxx223: starting org. apache. spark. deploy. worker. Worker, logging to /home/harli/cluster_13/spark - 1. 3. 0 - bin - hadoop2. 4/sbin/. . /logs/spark - harli - org. apache. spark. deploy. worker. Worker - 1 - wxx223. out
wxx215: starting org. apache. spark. deploy. worker. Worker, logging to /home/harli/cluster_13/spark - 1. 3. 0 - bin - hadoop2. 4/sbin/. . /logs/spark - harli - org. apache. spark. deploy. worker. Worker - 1 - wxx215. out

[harli@ wxx214spark - 1. 3. 0 - bin - hadoop2. 4] $ jps
30264NameNode
28542ResourceManager
26646TachyonMaster
30774 Master
30925 Jps
30461SecondaryNameNode
```

对应的停止命令为 stop - all，程序执行代码如下所示：

```
[harli@ wxx214 spark - 1. 3. 0 - bin - hadoop2. 4] $ ./sbin/stop - all. sh
wxx225: stopping org. apache. spark. deploy. worker. Worker
wxx223: stopping org. apache. spark. deploy. worker. Worker
wxx215: stopping org. apache. spark. deploy. worker. Worker
stopping org. apache. spark. deploy. master. Master

[harli@ wxx214 spark - 1. 3. 0 - bin - hadoop2. 4] $ jps
30264NameNode
28542ResourceManager
26646TachyonMaster
30692 Jps
30461SecondaryNameNode
```

八、查看当前集群启动进程情况

在 Master 节点，使用 jps 命令查看进程启动情况，输入命令后执行结果如下：

```
[harli@ wxx214 hadoop - 2. 6. 0] $ jps
26646TachyonMaster
22716 Jps
2609ResourceManager
2347SecondaryNameNode
2142NameNode
```

```
3187 Master
```

由上述执行结果可知，当前 Hadoop 的 NameNode，ResourceManager 等 Master 进程和 Spark 的 Master 进程部署在同一台机器上，即这里的命令行提示信息中的 wxx214 节点上。

其中各个服务支持的进程对应关系如下：

1）Hadoop 的 HDFS 服务：NameNode 进程。

2）Hadoop 的 Yarn 服务：ResourceManager 进程。

3）Spark 的 Master 服务：Master 进程。

在 Slaves 节点，使用 jps 命令查看进程启动情况，命令执行结果如下：

```
[harli@wxx215 hadoop-2.6.0]$ jps
29031 Jps
6243 Worker
5982NodeManager
5851DataNode
```

其中各个服务支持的进程对应关系如下：

1）Hadoop 的 HDFS 服务：DataNode 进程。

2）Hadoop 的 Yarn 服务：NodeManager 进程。

3）Spark 的 Slave 服务：Worker 进程。

2.2.2 交互式工具的启动

一、交互式工具的启动

进入 Spark 部署目录，基于单机模式启动 spark-shell 应用，在命令行提示符 $ 后面输入命令 ./bin/spark-shell --master。如下所示：

```
[harli@wxx215 spark-1.3.0-bin-hadoop2.4]$ ./bin/spark-shell --master
spark://192.168.70.214:7077
```

之后出现如下信息，这些信息中包含了交互式工具运行的环境信息（如是否带 Hive 等）以及预构建的一些实例（如预构建的 SparkContext 和 SQLContext 实例等）：

```
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
15/04/03 10:58:54 WARN util.NativeCodeLoader:Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/04/03 10:58:54 INFO spark.SecurityManager:Changing view acls to:harli
15/04/03 10:58:54 INFO spark.SecurityManager:Changing modify acls to:harli
15/04/03 10:58:54 INFO spark.SecurityManager:SecurityManager:authentication disabled;ui acls disabled;users with view permissions:Set(harli);users with modify permissions:Set(harli)
15/04/03 10:58:54 INFO spark.HttpServer:Starting HTTP Server
15/04/03 10:58:55 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/03 10:58:55 INFO server.AbstractConnector:Started SocketConnector@0.0.0.0:57536
15/04/03 10:58:55 INFO util.Utils:Successfully started service' HTTP class server' on port 57536.
Welcome to
```

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/
```

Using Scala version 2. 10. 4 (Java HotSpot(TM) Server VM, Java 1. 7. 0)
Type in expressions to have them evaluated.
Type:help for more information.

最后会生成 SparkContext 和 SQLContext 两个实例，这两个实例对应的名称分别为 sc 和 sqlContext。Spark 1. 3 版本在交互式工具 spark - shell 启动后，自动构建了这两个实例，通常用 SparkContext 代表 Driver Program。

在实例中将直接使用 sqlContext 进行操作，生成 SparkContext 和 SQLContext 两个实例见下面日志信息：

```
15/04/03 10:59:03 INFOrepl. SparkILoop:Created spark context. .
Spark context available as sc.
15/04/03 10:59:03 INFOrepl. SparkILoop:Created sql context (with Hive support). .
SQL context available assqlContext.
15/04/03 10:59:05 INFO cluster. SparkDeploySchedulerBackend: Registered executor: Actor [akka. tcp://sparkExecutor@ wxx225:18129/user/Executor# -1906560325] with ID 0
15/04/03 10:59:05 INFO cluster. SparkDeploySchedulerBackend: Registered executor: Actor [akka. tcp://sparkExecutor@ wxx223:52922/user/Executor# -1906560325] with ID 2
15/04/03 10:59:05 INFO storage. BlockManagerMasterActor: Registering block managerwxx225:18869 with 265. 0 MB RAM, BlockManagerId(0, wxx225, 18869)
15/04/03 10:59:05 INFO storage. BlockManagerMasterActor: Registering block managerwxx223:44620 with 265. 0 MB RAM, BlockManagerId(2, wxx223, 44620)
```

回车后就可以进入 Spark 应用程序的交互式界面，可以进行 Spark 应用程序的交互式开发、调试了。交互式界面上，输入命令的提示以 scala > 开头，执行输入的命令后会给出命令执行的日志和反馈信息。

当前交互式工具 spark - shell 使用了默认值 -- deploy - mode，即以客户端（Client）部署模式启动，因此，启动的 Driver Program 在提交节点上运行。通过启动节点的 http://driver:4040 地址可以打开 Driver 的 Web Interface 界面。

二、交互式工具的多次启动

（一）在 Spark 1. 3 之前的版本

在 Spark1. 3 之前的版本时，若要在相同节点上启动多个 Spark 应用程序时（特指部署模式为“Client”的情况下，Spark 应用程序的 Driver Program 在启动节点上运行），比如用交互式工具的方式多次启动 Spark 应用程序时，系统会报以下错误，如图 2. 3 所示：

这种情况下，后启动的 Spark 应用程序的 Driver 仍然使用了默认的 4040 端口，以致启动时地址被占用，进而最终 Spark 应用程序启动失败。

对应的解决方法如下：

1. 查看代码：SparkUI. scala

defgetUIPort(conf:SparkConf):Int = {conf. getInt("Spark. ui. port",SparkUI. DEFAULT_PORT)}

```
[root@n2 bin]# ./spark-submit --class PowerStreamStatis.RealToMinStatis /usr/l
al/yl/streamingTest.jar n1 9998 9993
connect success
15/01/27 10:50:51 WARN AbstractLifeCycle: FAILED SelectChannelConnector@0.0.0.0:
4040: java.net.BindException: 地址已在使用
java.net.BindException: 地址已在使用
        at sun.nio.ch.Net.bind0(Native Method)
        at sun.nio.ch.Net.bind(Net.java:444)
        at sun.nio.ch.Net.bind(Net.java:436)
        at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:
214)
        at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:74)
        at org.eclipse.jetty.server.nio.SelectChannelConnector.open(SelectChanne
lConnector.java:187)
        at org.eclipse.jetty.server.AbstractConnector.doStart(AbstractConnector.
```

图 2. 3　Driver Program 启动时地址被占用的界面

2. 修改配置，添加 spark. ui. port 属性。如提交时，添加：－－conf　"spark. ui. port"　=4041 即可。

注意：Spark 的属性配置可以通过三种方式进行设置：

1）可以在程序部署时的提交命令的－－conf 选项中通过设置 Spark 的配置文件（如 conf/spark－defalut. conf）进行设置。

2）可以在 Spark 的环境变量文件中的 java 启动选项（如在 conf/spark－env. sh 文件中的环境变量 Spark _DAEMON_JAVA_OPTS）里添加 Spark 的属性配置。

3）可以通过指定配置文件等方式进行设置。

如果 spark. ui. port 的值设置为 0，系统就会随机选取一个端口号。参考源码：java. net. ServerSocket：ServerSocket 对象的绑定端口为 0，getLocalPort 方法返回一个随机的端口（这类端口被称为匿名端口）。

（二）Spark 1. 3 版本

Spark1. 3 版本可在一个节点上多次启动 Spark 应用程序，如多次运行交互式工具 spark－shell。具体方式是，Spark1. 3 在已启动应用程序的计算机上再次启动应用时，如果启动应用的命令没有指定 Driver Progam 的 Web Interface 的端口号的话，启动的应用会先从默认的端口 4040 开始启动，由于该端口已经被占用，因此首次启动时会报错，如下面的日志所示：

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) Server VM,Java 1.7.0)
Type in expressions to have them evaluated.
Type:help for more information.
15/04/03 15:55:23 INFO spark.SparkContext:Running Spark version 1.3.0
```

```
15/04/03 15:55:23 INFO spark.SecurityManager:Changing view acls to:harli
15/04/03 15:55:23 INFO spark.SecurityManager:Changing modify acls to:harli
15/04/03 15:55:23 INFO spark.SecurityManager:SecurityManager:authentication disabled;ui acls disabled;users with view permissions:Set(harli);users with modify permissions:Set(harli)
15/04/03 15:55:23 INFO slf4j.Slf4jLogger:Slf4jLogger started
15/04/03 15:55:23 INFORemoting:Starting remoting
15/04/03 15:55:24 INFORemoting:Remoting started;listening on addresses:[akka.tcp://sparkDriver@wxx215:52994]
15/04/03 15:55:24 INFO util.Utils:Successfully started service'sparkDriver' on port 52994.
15/04/03 15:55:24 INFO spark.SparkEnv:Registering MapOutputTracker
15/04/03 15:55:24 INFO spark.SparkEnv:Registering BlockManagerMaster
15/04/03 15:55:24 INFO storage.DiskBlockManager:Created local directory at /tmp/spark-201897f2-00e5-4ffa-b346-f51642a6861f/blockmgr-d73fcef7-aea5-4bb9-8b0a-96e01a50580d
15/04/03 15:55:24 INFO storage.MemoryStore:MemoryStore started with capacity 265.0 MB
15/04/03 15:55:24 INFO spark.HttpFileServer:HTTP File server directory is /tmp/spark-bd5925eb-1619-499a-ac96-51827ae9b02d/httpd-7c435146-3962-48dc-9414-bbaf1ebedc00
15/04/03 15:55:24 INFO spark.HttpServer:Starting HTTP Server
15/04/03 15:55:24 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/03 15:55:24 INFO server.AbstractConnector:Started SocketConnector@0.0.0.0:61139
15/04/03 15:55:24 INFO util.Utils:Successfully started service'HTTP file server' on port 61139.
15/04/03 15:55:24 INFO spark.SparkEnv:Registering OutputCommitCoordinator
15/04/03 15:55:24 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/03 15:55:24 WARN component.AbstractLifeCycle:FAILED SelectChannelConnector@0.0.0.0:4040:java.net.BindException:地址已在使用
java.net.BindException:地址已在使用
        at sun.nio.ch.Net.bind0(Native Method)
```

当默认端口4040绑定失败后，提交应用程序会递增端口号（4041、4042、……），尝试重新启动Spark应用，直到找到一个可用的端口号为止，这时候，启动Driver Program成功，进入交互式界面。下列代码所示为已经成功绑定了4041端口号，成功启动Spark应用，代码执行结果如下：

```
15/04/03 15:55:24 INFO handler.ContextHandler:stopped o.s.j.s.ServletContextHandler{/jobs,null}
15/04/03 15:55:24 WARN util.Utils:Service'SparkUI' could not bind on port 4040. Attempting port 4041.
15/04/03 15:55:24 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/03 15:55:24 INFO server.AbstractConnector:Started SelectChannelConnector@0.0.0.0:4041
15/04/03 15:55:24 INFO util.Utils:Successfully started service'SparkUI' on port 4041.
15/04/03 15:55:24 INFO ui.SparkUI:Started SparkUI at http://wxx215:4041
15/04/03 15:55:24 INFO client.AppClient $ ClientActor:Connecting to master akka.tcp://sparkMaster@192.168.70.214:7077/user/Master...
15/04/03 15:55:25 INFO cluster.SparkDeploySchedulerBackend:Connected to Spark cluster with app ID app-20150403155539-0004
15/04/03 15:55:25 INFO netty.NettyBlockTransferService:Server created on 33612
15/04/03 15:55:25 INFO storage.BlockManagerMaster:Trying to register BlockManager
15/04/03 15:55:25 INFO storage.BlockManagerMasterActor:Registering block managerwxx215:33612 with 265.0 MB RAM,BlockManagerId(<driver>,wxx215,33612)
15/04/03 15:55:25 INFO storage.BlockManagerMaster:Registered BlockManager
```

```
15/04/03 15:55:26 INFO scheduler. EventLoggingListener:Logging events tohdfs://wxx214:9000/logs/app-20150403155539-0004
15/04/03 15:55:26 INFO cluster. SparkDeploySchedulerBackend:SchedulerBackend is ready for scheduling beginning after reached minRegisteredResourcesRatio:0.0
15/04/03 15:55:26 INFOrepl. SparkILoop:Created spark context..
Spark context available as sc.
15/04/03 15:55:26 INFOrepl. SparkILoop:Created sql context (with Hive support)..
SQL context available assqlContext.
```

三、Driver Program 的 Web Interface 界面

Driver Program 的 Web Interface 界面用于查看该 Driver Program 的信息，包括 Driver Program 的 Jobs 信息、Stages 信息、Storage 信息、Environment 信息和 Executors 信息。

（一）打开 Web Interface 页面

Web Interface 地址：http://driverhost:4040/jobs/，界面内容如图 2.4 所示。

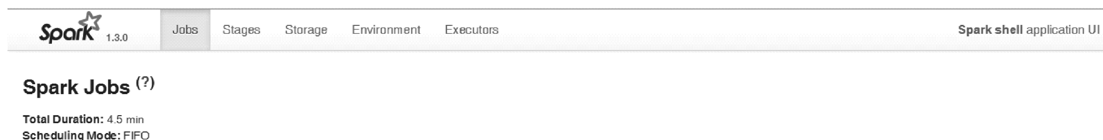


图 2.4　Driver Program 监控界面

其中，driverhost 为交互式工具启动节点的 hostname。

Web Interface 界面包含 Spark 版本信息，以及当前 Driver Program 中的 Jobs、Stages、Storage、Environment 以及 Executors 信息。

注意：当使用 hostname 时，需要保证已经在系统的 hosts 配置文件中添加了 hostname 与 IP 地址的对应关系。在 Linux 系统上，hosts 的配置方法如下：

1. 使用 vim 工具打开 hosts 文件：vim /etc/hosts。
2. 在文件中添加配置的 hostname 与 IP 的映射关系，比如：

```
[harli@cluster04Spark]$ cat /etc/hosts
#127.0.0.1      localhost localhost.localdomain localhost4 localhost4.localdomain4
#::1            localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.242.134    cluster01
192.168.242.135    cluster04
192.168.242.136    cluster05
192.168.242.137    cluster06
```

（二）Web Interface 界面内容解析

Driver Program 的 Web Interface 界面中各个 Tab 页包含的详细信息如下：

1. Jobs 页。打开 driver program 监控界面时的默认页，如图 2.5 所示。

在 Jobs 页中，包含了当前 Driver Program 运行持续的时间，当前 Jobs 的调度模式（默认调度模式为 FIFO）以及全部 Jobs 的具体信息。

注意：由于当前未提交任何 Job，界面的 Jobs 内容为空。

2. Stages 页。显示当前 Driver Program 所有 Jobs 的 Stages 信息，包含的具体内容如



图 2.5　Driver Program 监控界面的 Jobs 页

图 2.6所示。



图 2.6　Driver Program 监控界面的 Stages 页

在 Stages 页面，包含了当前 Driver Program 运行持续的时间，当前 Stages 的调度模式（默认调度模式为 FIFO）以及全部 Jobs 的 Stages 的具体信息。

3. Storage 页。显示当前 Driver Program 所有 Jobs 的 Storage 信息，包含的具体内容如图 2.7所示。

Spark 1.3.0　Jobs　Stages　Storage　Environment　Executors　Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk

图 2.7　Driver Program 监控界面的 Storage 页

在 Stages 页面，包含了当前 Driver Program 的存储信息，存储信息包含了 RDD 的名字、存储级别、缓存的分区数、缓存分区占全部分区的比例、缓存在内存的数据大小、缓存在 Tachyon 中的数据大小、存储在磁盘的数据大小。

4. Environment 页。显示当前 Driver Program 的 Environment 信息，包含的具体内容如图 2.8所示。

在 Environment 页面，包含了当前 Driver Program 的环境配置信息，具体包含四大类：Runtime Information、Spark Properties、System Properties 以及 Classpath Entries。

5. Executors 页。显示当前 Driver Program 的 Executors 信息，包含的具体内容如图 2.9 所示。

在 Environment 页面，包含了当前 Spark 应用程序用到的 Executors 信息，包含当前 Spark 应用程序分配的总内存大小，使用的磁盘大小以及 Executor 的具体信息。

Executor 的信息包含：Executor 的 ID、执行的地址（host：port）、RDD 的分块数（Blocks）、内存使用大小、磁盘使用大小、当前 Active 状态的任务数、失败的任务数、总的任务数等信息。

由于当前 Spark 应用程序仅启动了 Driver Program，故只分配了一个 Executor，并且 Executor ID 为 <driver>。

Spark 1.3.0　Jobs　Stages　Storage　Environment　Executors　Spark shell application UI

Environment

Runtime Information

Name	Value
Java Home	D:\Java\jdk1.7.0_71\jre
Java Version	1.7.0_71 (Oracle Corporation)
Scala Version	version 2.10.4

Spark Properties

Name	Value
spark.app.id	local-1427296029170
spark.app.name	Spark shell
spark.driver.host	lenovo-PC
spark.driver.port	49735
spark.executor.id	<driver>
spark.fileserver.uri	http://192.168.1.103:49736
spark.jars	
spark.master	local[*]
spark.repl.class.uri	http://192.168.1.103:49684

图 2.8　Driver Program 监控界面的 Environment 页

Spark 1.3.0　Jobs　Stages　Storage　Environment　Executors　Spark shell application UI

Executors (1)

Memory: 0.0 B Used (265.4 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Thread Dump
<driver>	localhost:49758	0	0.0 B / 265.4 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	Thread Dump

图 2.9　Driver Program 监控界面的 Executors 页

2.2.3　文本数据的 ETL 案例实践与解析

一般情况下，公司内部需要处理的文件都是以文本文件的格式保存，包括公司的网站访问日志文件、公司系统运行监控日志文件等，在收集到这些文本文件后，需要进行各种数据处理，包括数据的抽取－转换－装载（Extract－Transform－Load，ETL）、数据统计、数据挖掘，以及为后续的数据呈现和为决策而提供的数据持久化。这里给出的文本文件的处理过程，包含了对外部文件的加载，对文件数据的转换、过滤、各种数据统计，以及处理结果的存储。

一、准备要处理的文件

1. 在 Hadoop 部署目录下输入查询、创建目录命令，以查看文件系统：

```
[harli@wxx214 hadoop-2.6.0]$ ./bin/hdfs dfs -ls /
[harli@wxx214 hadoop-2.6.0]$ ./bin/hdfs dfs -mkdir /user
[harli@wxx214 hadoop-2.6.0]$ ./bin/hdfs dfs -ls /
Found1 items
drwxr-xr-x   - hdfs   supergroup         0 2015-01-22 10:54 /user
```

除了以命令行方式查看检查文件外，还可以登录 Hadoop 的 Web Interface（http://name-

node:50070）页面查看文件，如图 2. 10 所示。其中 namenode 为启动 NameNode 进程的节点，当前环境下的节点地址为 192. 168. 70. 214。

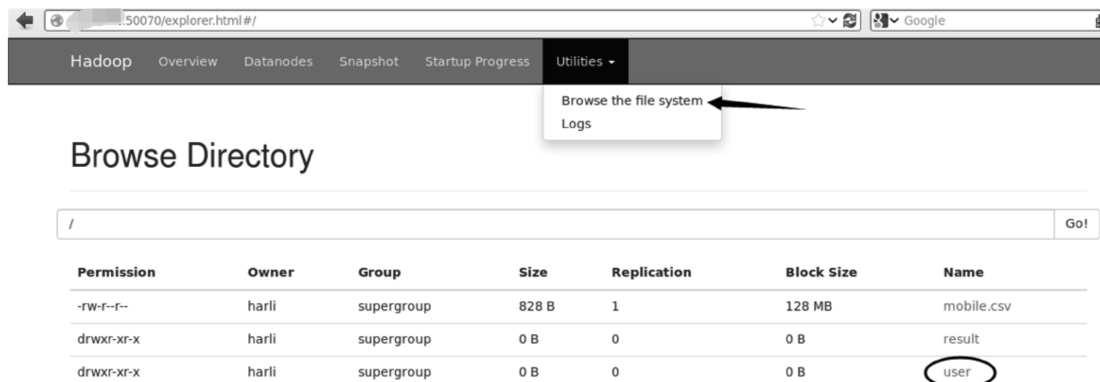


图 2. 10　Hadoop 的文件系统查看界面

2. 将本地文件 README. md 上传到 Hadoop：

```
[harli@ wxx214 hadoop －2. 6. 0] $ ./bin/hdfs dfs －put /home/harli/cluster_13/Spark －1. 3. 0 －bin －hadoop2. 4/README. md /user
[harli@ wxx214 hadoop －2. 6. 0] $ ./bin/hdfs dfs －ls / user
Found1 items
－rw －r －－r －－    3harli supergroup        3629 2015 －03 －26 10:36 /user/README. md
```

登录 Web Interface（http://namenode:50070）页面，查看文件，文件已经添加到/user 目录，添加后的文件系统如图 2. 11 所示。

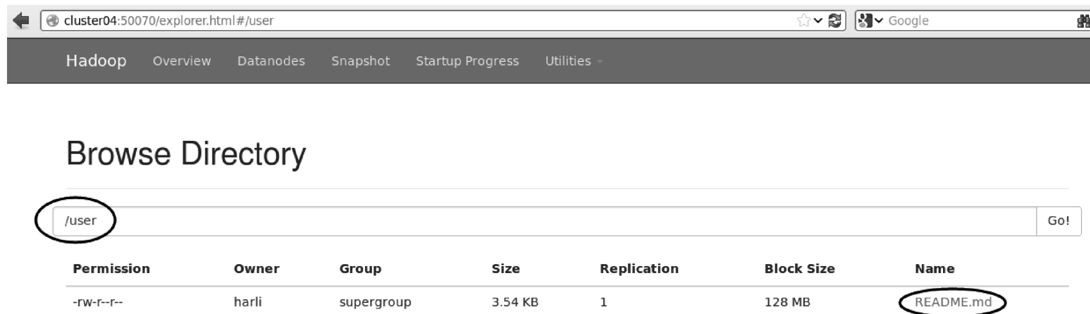


图 2. 11　Hadoop 的文件系统界面界面上传结果

二、启动交互式界面

1. 基于集群模式启动 spark －shell 应用：

```
[harli@ wxx215 Spark －1. 3. 0 －bin －hadoop2. 4] $ ./bin/spark －shell －－master Spark://192. 168. 70. 214:7077
```

其中，Spark://192. 168. 70. 214:7077 为当前集群的 MasterURL。需要注意的是，MasterURL 必须和 Spark 界面（http://master:8080）上一致，hostname 和 IP 之间不能互相替换。



启动交互式 spark - shell 工具时，交互式 spark - shell 工具默认构建 SparkContext 和 SQLContext 实例，分别为 sc 和 sqlContext：

```
15/04/03 02:50:41 INFOBlockManagerMaster:Registered BlockManager
15/04/03 02:50:42 INFOSparkILoop:Created spark context..
Spark context available as sc.
15/04/03 02:50:43 INFOSparkILoop:Created sql context (with Hive support)..
SQL context available assqlContext.
```

2. 加载文件构建 RDD，输入命令：val textFile = sc. textFile("/user/README. md"),执行结果如下所示：

```
scala > val textFile = sc. textFile("/user/README. md")
15/04/03 11:22:02 INFO storage. MemoryStore: ensureFreeSpace(206436) called with curMem = 724503,maxMem = 277842493
15/04/03 11:22:02 INFO storage. MemoryStore:Block broadcast_7 stored as values in memory (estimated size 201.6 KB,free:264.1 MB)
15/04/03 11:22:02 INFO storage. MemoryStore: ensureFreeSpace(31717) called with curMem = 930939,maxMem = 277842493
15/04/03 11:22:02 INFO storage. MemoryStore:Block broadcast_7_piece0 stored as bytes in memory (estimated size 31.0 KB,free:264.1 MB)
15/04/03 11:22:02 INFO storage. BlockManagerInfo:Added broadcast_7_piece0 in memory on wxx215:21048 (size:31.0 KB,free:264.8 MB)
15/04/03 11:22:02 INFO storage. BlockManagerMaster:Updated info of block broadcast_7_piece0
15/04/03 11:22:02 INFO spark. SparkContext:Created broadcast 7 from textFile at <console>:24
textFile:org. apache. spark. rdd. RDD[String] = /user/README. md MapPartitionsRDD[13] at textFile at <console>:24
```

使用 SparkContext 的 textFile 来加载文件，加载后的 textFile，其组成元素为文件的每一行记录，元素类型为 String。

注意：当 Spark 基于 Hadoop 的 HDFS 存储系统时，由于 HDFS 默认使用的 Scheme 是 hdfs，因此，“/user/README. md”等同于 hdfs 的“fs. defaultFS + ‘/user/README. md’”，在本集群中对应为“hdfs://wxx214:9000/user/README. md”。

Hadoop 集群的 core - site. xml 文件中的相关配置：

```
<property>
<name>fs. defaultFS</name>
<value>hdfs://wxx214:9000</value>
<description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation.  The
  uri's scheme determines the config property (fs. SCHEME. impl) naming
  the FileSystem implementation class.  The uri's authority is used to
  determine the host,port,etc. for afilesystem. </description>
</property>
```

该信息也可以在 Hadoop 的 Web Interface（http://namenode:50070）页面上找到。

3. 修改日志等级，去除多余的日志信息：

```
import org. apache. log4j. {Level,Logger}
```

```
Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
```

这里将日志等级设置为 Level.WARN，即只输出 Level.WARN 或更高级别的日志信息。

4. 使用 take 函数获取 textFile 的前 n 个元素，执行结果如下：

```
scala > textFile.take(1)
15/04/03 11:28:11 INFOmapred.FileInputFormat:Total input paths to process:1
res1:Array[String] = Array(# Apache Spark)
```

take 函数是将 RDD 的前 n 个元素抽取出来作为 Array 返回。它会先扫描第一个分区的元素个数，并用这个数评估获取 n 个数据还需要扫描的分区个数。textFile.take（1）获取了当前第一个元素，即加载文件的第一行：# Apache Spark。

5. 使用 first 方法获取 textFile 的第一个元素，执行结果如下：

```
scala > textFile.first
res2:String = # Apache Spark
```

RDD 的 first 方法实际上是调用了 take(1)方法，textFile.first 获取了第一个元素，即加载文件的第一行：# Apache Spark。

需要注意的是，first 和 take 方法的返回值的类型不同，first 返回 String，take 返回的是 Array[String]。

6. 运用 filter 方法指定元素的过滤条件，并获取过滤的结果，执行结果如下：

```
scala > vallinesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark:org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at filter at <console
>:24
```

RDD 的 filter 方法是对 RDD 中的数据单元进行过滤。如果数据单元在 filter 方法指定的函数值执行后返回 true，则当前被验证的 RDD 元素会被过滤出来，最后构建的 RDD 只包含符合过滤条件的元素。

其中，参数 line 对应于 RDD 的元素，也就是 textFile 加载的文件的行，textFile.filter(line => line.contains("Spark"))表示过滤出包含了"Spark"字符串的每一行。

7. 运用 collect 方法获取 RDD 的内容，执行结果如下：

```
scala > linesWithSpark.collect
res3:Array[String] = Array(# Apache Spark,Spark is a fast and general cluster computing system for Big
Data. It provides,rich set of higher - level tools including Spark SQL for SQL and structured,and Spark
Streaming for stream processing.,You can find the latest Spark documentation,including a programming,#
# Building Spark,Spark is built using [Apache Maven](http://maven.apache.org/).,To build Spark
and its example programs,run:,["Building Spark"](http://spark.apache.org/docs/latest/building -
spark.html).,The easiest way to start using Spark is through the Scala shell:,Spark also comes with sev-
eral sample programs in the' examples' directory.,"./bin/run - example SparkPi"," MASTER =
spark://host:7077 ./bin/run - example SparkPi",Testing first requires [building Spark](#bu...
```

collect 方法会将 RDD 的元素返回到 Driver 端，放到一个数组中，需要注意的是，如果 RDD 的数据量太大的话，collect 方法会消耗很大的内存。

2.2.4 文本数据的初步统计案例实践与解析

在对上一节的文本数据（即 README.md 文件的数据）进行初步的 ETL 操作之后，就

可以开始进行数据统计、挖掘等操作了，这里主要介绍数据的一些统计操作，以及为了加快统计效率而使用的持久化操作。

1. 使用 count 方法统计 textFile 的元素个数，即文件行数，执行结果如下：



```
scala > textFile. count
res6:Long = 98
```

可以看到文件的行数为 98。

2. 获取单词个数最多的行的单词个数。

```
scala > textFile. map( line => line. split(" "). size). reduce( (a,b) => if (a > b) a else b)
res4:Int = 14
```

其中：

1）map 方法是针对 RDD 的每个元素执行 map 函数，并返回处理后的新 RDD。map（line => line. split（""）. size）操作是对每个元素按空格进行 split 后，再获取 split 得到的数组的长度，其结果就是将 RDD 的元素转换为元素包含的单词的个数。

2）reduce 是先对 RDD 每个分区的数据进行归并操作，最终对每个分区数据的归并结果再次进行归并。reduce（（a，b） => if（a > b）a else b）是获取 map 后 RDD 元素的最大值。这里是比较每行包含的单词个数。

查看 map 后每行数据的单词个数，单词个数最多的为 14，执行结果如下所示：

```
scala > textFile. map( line => line. split(" "). size). collect
res5:Array[Int] = Array(3,1,14,12,11,12,10,6,1,1,1,1,3,1,10,6,3,8,1,3,1,6,8,1,8,1,13,10,
2,1,4,1,12,1,5,1,8,1,8,1,4,1,11,1,5,0,10,1,6,1,3,1,11,11,1,6,1,6,1,12,12,11,13,14,3,
1,7,1,13,1,3,1,10,4,1,5,1,7,4,1,6,1,13,11,13,1,7,4,12,10,4,12,1,1,2,1,6,12)
```

改用 Math 方法获取最大值，执行结果如下所示：

```
scala > import java. lang. Math
import java. lang. Math
scala > textFile. map( line => line. split(" "). size). reduce( (a,b) => Math. max(a,b))
res6:Int = 14
```

Scala 编译后也是 . class 文件，在 JVM 中运行，因此可以调用同样编译成 . class 文件的 java 类，即可以使用已有的 java 类库，这里使用了 java. lang. Math 中的 Math. max 方法。

3. 实现 WordCounts（单词统计），并获取统计结果，输入命令，查看命令结果：

```
scala > val wordCounts = textFile. flatMap ( line => line. split ( " " ) ) . map ( word => ( word, 1 ) )
. reduceByKey( (a,b) => a + b)
wordCounts:org. apache. Spark. rdd. RDD[(String,Int)] = ShuffledRDD[8] at reduceByKey at <con-
sole > :25
```

其中：

1）flatMap(line => line. split(" "))，flatMap 操作，相当于先将元素 map 一个数据集，然后把数据集 flap，即扁平化处理。案例中，是将 textFile 的元素，即输入文件的每一行，按空格（" "）进行拆分，得到单词数组，最后将数组进行扁平化后 textFile 的元素变为单词字符串。

2）map 方法在这里是将 RDD 的每个元素转换为一个二元组，二元组包含一个单词和初

始统计值 1。

3）reduceByKey 方法和 reduce 方法一样，只是针对相同 Key 值的 value 进行归并操作。在这里针对相同元素，即相同单词进行归并（即求和），最后得出单词的个数统计值。

通过 collect 方法查看 wordCounts 的内容，执行结果如下：

```
scala > wordCounts. collect
res7:Array[(String,Int)] = Array((package,1),(this,1),(Because,1),(Python,2),(cluster. ,1),
(its,1),([run,1),(general,2),(YARN,,1),(have,1),(pre - built,1),(locally. ,1),(changed,
1),(locally,2),(sc. parallelize(1,1),(only,1),(several,1),(This,2),(basic,1),(first,1),(docu-
mentation,3),(Configuration,1),(learning,,1),(graph,1),(Hive,2),(["Specifying,1),("yarn -
client",1),(page](http://spark. apache. org/documentation. html),1),([params]` . ,1),(applica-
tion,1),([project,2),(prefer,1),(SparkPi,2),( < http://spark. apache. org/ > ,1),(engine,1),
(version,1),(file,1),(documentation,,1),(MASTER,1),(example,3),(are,1),(systems. ,1),
(params,1),(scala > ,1),(provides,1),(refer,2),(configure,1),(Interactive,2),(distribution. ,
1),(can,6),(build,3),(when,1),(Apache,1),...
```

4. 对每行单词个数求 top，执行结果如下：

```
scala > textFile. map(line => line. split(" "). size). top(3)
res4:Array[Int] = Array(14,14,13)
```

这里将 textFile 的每条记录，即文件的每一行，map 为单词个数，然后求出单词个数排前三的个数值。

5. 测试 textFile 的缓存。为了测试缓存时间的变化，修改日志等级进行调试，语句如下：

```
Logger. getLogger("org. apache. spark"). setLevel(Level. INFO)
```

测试过程：先执行 textFile. count 统计元素个数，然后缓存 textFile，并用 collect 触发，触发后再次执行 textFile. count 命令，比对缓存前后，元素个数统计所消耗的时间。依次输入命令：

```
scala > textFile. count
scala > textFile. cache
scala > textFile. collect
scala > textFile. count
```

其中：

1）cache 方法：将 RDD 缓存到内存中，即 persist(StorageLevel. MEMORY_ONLY)的缩写，是一个 lazy 操作（一旦存储等级被改变，必须先调用 unpersist 去除后才能重新设置新的存储等级）。

2）collect 方法：这是一个 Action 操作。会获取 RDD 的全部元素，并转换为 Scala Array 返回给 Driver Program，collect 方法中还可以添加一个偏函数对返回的数据进行预处理，处理得到的类型可以和 RDD 的元素类型不同。通常情况下，在 RDD 数据量大时应避免使用该方法，避免在 Driver Program 中内存不足以装载全部元素而导致的内存溢出问题。

如图 2. 12 所示，缓存前，统计时间为 0. 085439s，缓存后的统计时间为 0. 047926s。

```
scala>  textFile.count
15/04/03 11:41:38 INFO spark.SparkContext: Starting job: count at <console>:26
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Got job 16 (count at <console>:26) with 2 output partiti
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Final stage: Stage 17(count at <console>:26)
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Parents of final stage: List()
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Missing parents: List()
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Submitting Stage 17 (/user/README.md MapPartitionsRDD[1]
15/04/03 11:41:38 INFO storage.MemoryStore: ensureFreeSpace(2640) called with curMem=281838, maxMem=277
15/04/03 11:41:38 INFO storage.MemoryStore: Block broadcast_18 stored as values in memory (estimated si
15/04/03 11:41:38 INFO storage.MemoryStore: ensureFreeSpace(1929) called with curMem=284478, maxMem=277
15/04/03 11:41:38 INFO storage.MemoryStore: Block broadcast_18_piece0 stored as bytes in memory (estima
15/04/03 11:41:38 INFO storage.BlockManagerInfo: Added broadcast_18_piece0 in memory on wison215:12601
15/04/03 11:41:38 INFO storage.BlockManagerMaster: Updated info of block broadcast_18_piece0
15/04/03 11:41:38 INFO spark.SparkContext: Created broadcast 18 from broadcast at DAGScheduler.scala:83
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Submitting 2 missing tasks from Stage 17 (/user/README.m
15/04/03 11:41:38 INFO scheduler.TaskSchedulerImpl: Adding task set 17.0 with 2 tasks
15/04/03 11:41:38 INFO scheduler.TaskSetManager: Starting task 0.0 in stage 17.0 (TID 32, wison225, NOD
15/04/03 11:41:38 INFO scheduler.TaskSetManager: Starting task 1.0 in stage 17.0 (TID 33, wison215, NOD
15/04/03 11:41:38 INFO storage.BlockManagerInfo: Added broadcast_18_piece0 in memory on wison225:6016 (
15/04/03 11:41:38 INFO storage.BlockManagerInfo: Added broadcast_18_piece0 in memory on wison215:15963
15/04/03 11:41:38 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 17.0 (TID 32) in 49 ms on w
15/04/03 11:41:38 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 17.0 (TID 33) in 64 ms on w
15/04/03 11:41:38 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 17.0, whose tasks have all complete
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Stage 17 (count at <console>:26) finished in 0.065 s
15/04/03 11:41:38 INFO scheduler.DAGScheduler: Job 16 finished: count at <console>:26, took 0.085439 s
res25: Long = 98

scala>  textFile.cache
res26: textFile.type = /user/README.md MapPartitionsRDD[1] at textFile at <console>:21
```

……

```
scala> textFile.collect
15/04/03 11:41:46 INFO storage.BlockManager: Removing broadcast 17
15/04/03 11:41:46 INFO storage.BlockManager: Removing block broadcast_17_piece0
15/04/03 11:41:46 INFO storage.MemoryStore: Block broadcast_17_piece0 of size 1936 droppe
15/04/03 11:41:46 INFO storage.BlockManagerInfo: Removed broadcast_17_piece0 on wison215:
15/04/03 11:41:46 INFO storage.BlockManagerMaster: Updated info of block broadcast_17_pie
```

……

```
documentation.html), and [project wiki](https://cwiki.apache.org/confluence/display/SPARK)., This README
scala>  textFile.count
15/04/03 11:41:50 INFO spark.SparkContext: Starting job: count at <console>:26
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Got job 18 (count at <console>:26) with 2 output partitio
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Final stage: Stage 19(count at <console>:26)
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Parents of final stage: List()
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Missing parents: List()
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Submitting Stage 19 (/user/README.md MapPartitionsRDD[1]
15/04/03 11:41:50 INFO storage.MemoryStore: ensureFreeSpace(2640) called with curMem=272693, maxMem=2778
15/04/03 11:41:50 INFO storage.MemoryStore: Block broadcast_20 stored as values in memory (estimated siz
15/04/03 11:41:50 INFO storage.MemoryStore: ensureFreeSpace(1936) called with curMem=275333, maxMem=2778
15/04/03 11:41:50 INFO storage.MemoryStore: Block broadcast_20_piece0 stored as bytes in memory (estimat
15/04/03 11:41:50 INFO storage.BlockManagerInfo: Added broadcast_20_piece0 in memory on wison215:12601 (
15/04/03 11:41:50 INFO storage.BlockManagerMaster: Updated info of block broadcast_20_piece0
15/04/03 11:41:50 INFO spark.SparkContext: Created broadcast 20 from broadcast at DAGScheduler.scala:839
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Submitting 2 missing tasks from Stage 19 (/user/README.md
15/04/03 11:41:50 INFO scheduler.TaskSchedulerImpl: Adding task set 19.0 with 2 tasks
15/04/03 11:41:50 INFO scheduler.TaskSetManager: Starting task 0.0 in stage 19.0 (TID 36, wison223, PROC
15/04/03 11:41:50 INFO scheduler.TaskSetManager: Starting task 1.0 in stage 19.0 (TID 37, wison225, PROC
15/04/03 11:41:50 INFO storage.BlockManagerInfo: Added broadcast_20_piece0 in memory on wison223:31143 (
15/04/03 11:41:50 INFO storage.BlockManagerInfo: Added broadcast_20_piece0 in memory on wison225:6016 (s
15/04/03 11:41:50 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 19.0 (TID 37) in 34 ms on wi
15/04/03 11:41:50 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 19.0 (TID 36) in 38 ms on wi
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Stage 19 (count at <console>:26) finished in 0.038 s
15/04/03 11:41:50 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 19.0, whose tasks have all completed
15/04/03 11:41:50 INFO scheduler.DAGScheduler: Job 18 finished: count at <console>:26, took 0.047926 s
res28: Long = 98
```

图 2.12 RDD 缓存及其触发过程的日志截图

对 RDD 进行缓存后，可以极大提高 RDD 的统计效率，减少多次对 RDD 统计时重新计算 RDD 的开销。

2.2.5 文本数据统计结果的持久化案例实践与解析

在对文本数据进行初步的 ETL 操作，以及对一些指标信息进行统计之后，通常需要将统计结果持久化，以便用于界面呈现以及向管理层提供决策所需信息。

1. 修改日志等级，保存文件，输入以下命令

```
scala > Logger. getLogger( "org. apache. Spark" ). setLevel( Level. WARN)
scala > wordCounts. saveAsTextFile( "/user/result0" )
```

保存为文本文件，由于当前是基于 HDFS 的文件系统，因此默认使用的是 HDFS 的 scheme，会保存到 HDFS 系统上。

使用 hdfs 命令查看：

```
[harli@ wxx214 hadoop -2. 6. 0] $ ./bin/hdfs dfs -ls /user
Found 8 items
-rw-r--r--    3harli supergroup     3629 2015 -03 -26 10:36 /user/README. md
drwxr-xr-x     - harli supergroup      0 2015 -04 -03 10:48 /user/harli
drwxr-xr-x     - harli supergroup      0 2015 -04 -01 13:26 /user/hive
drwxr-xr-x     - harli supergroup      0 2015 -03 -26 11:43 /user/objresult1
drwxr-xr-x     - harli supergroup      0 2015 -04 -03 11:48 /user/result0
```

查看 Web Interface（http://namenode:50070）页面，文件已经保存到/user/result。目录下，保存后的文件系统如图 2. 13 所示。

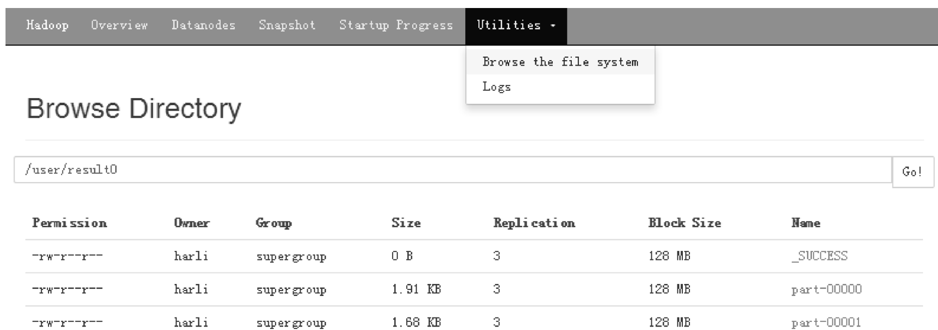


图 2. 13　Hadoop 文件系统上的保存结果界面

单击文件后可打开下载界面，界面包含的文件内容如图 2. 14 所示。

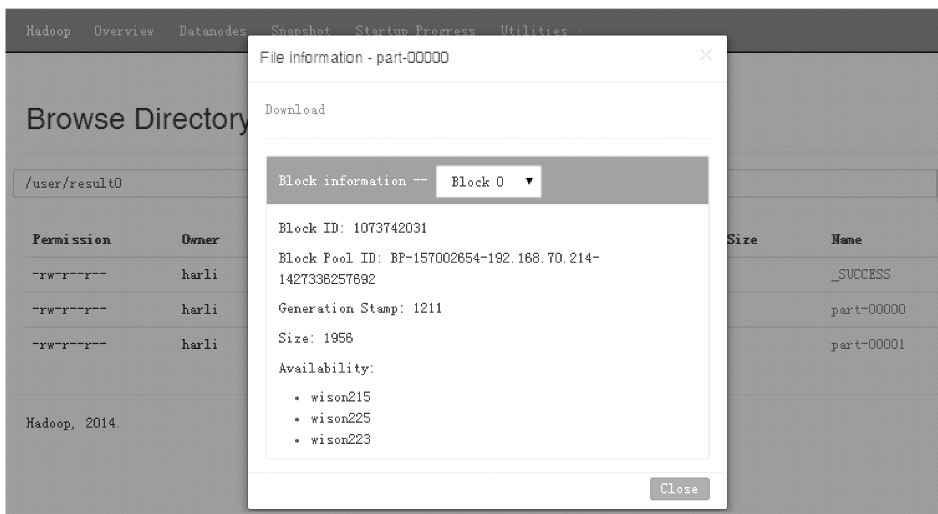


图 2. 14　Hadoop 文件系统的文件下载界面

2. 指定 scheme 保存文件，输入命令

```
scala > wordCounts. saveAsTextFile( " hdfs://wxx214:9000/user/result00" )
```

使用 hdfs 命令查看：

```
[harli@ wxx214 hadoop - 2. 6. 0] $ ./bin/hdfs dfs - ls /user
Found 9 items
- rw - r - - r - -    3harli supergroup      3629 2015 - 03 - 26 10:36 /user/README. md
drwxr - xr - x         - harli supergroup      0 2015 - 04 - 03 10:48 /user/harli
drwxr - xr - x         - harli supergroup      0 2015 - 04 - 01 13:26 /user/hive
drwxr - xr - x         - harli supergroup      0 2015 - 03 - 26 11:43 /user/objresult1
drwxr - xr - x         - harli supergroup      0 2015 - 04 - 03 11:48 /user/result0
drwxr - xr - x         - harli supergroup      0 2015 - 04 - 03 11:52 /user/result00
```

查看 Web Interface（http://namenode:50070）页面，文件已经保存到/user/result00 目录下，保存后的文件系统如图 2. 15 所示。

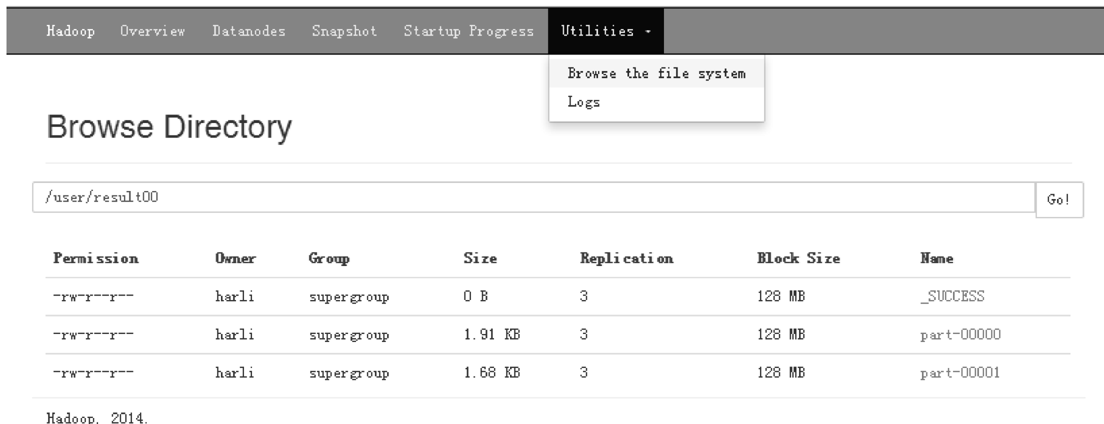


图 2. 15 Hadoop 文件系统上的保存结果界面

2. 2. 6 RDD 的 Lineage 关系的案例与源码解析

RDD 作为 Spark 的基本计算单元，是 Spark 的一个最核心的抽象概念，可以通过一系列算子进行操作，包括 Transformation 和 Action 两种算子操作。本节针对前面的案例进行详细解析，主要从源码、界面监控信息等方面进行详细解析。

接下来采用 Spark Shell 交互式工具运行 Spark 应用程序，应用程序的运行方式参见章节 2. 1Spark 应用程序部署。

在此先介绍一些代码阅读技巧，常用的一些代码操作技巧包括：

1）在 Shell 交互界面上，可以通过按【TAB】键自动补全代码。

2）IntellijIDEA 中：按住【CTRL】键，同时单击某个符号（如类、方法等），可以跳到该符号的定义；或鼠标定位于该符号，然后按【F4】键或【Ctrl + B】组合键，跳转到该符号的定义；或在符号处单击右键，在弹出上下文菜单中，选择 Go To→Declaration，如图 2. 16所示。

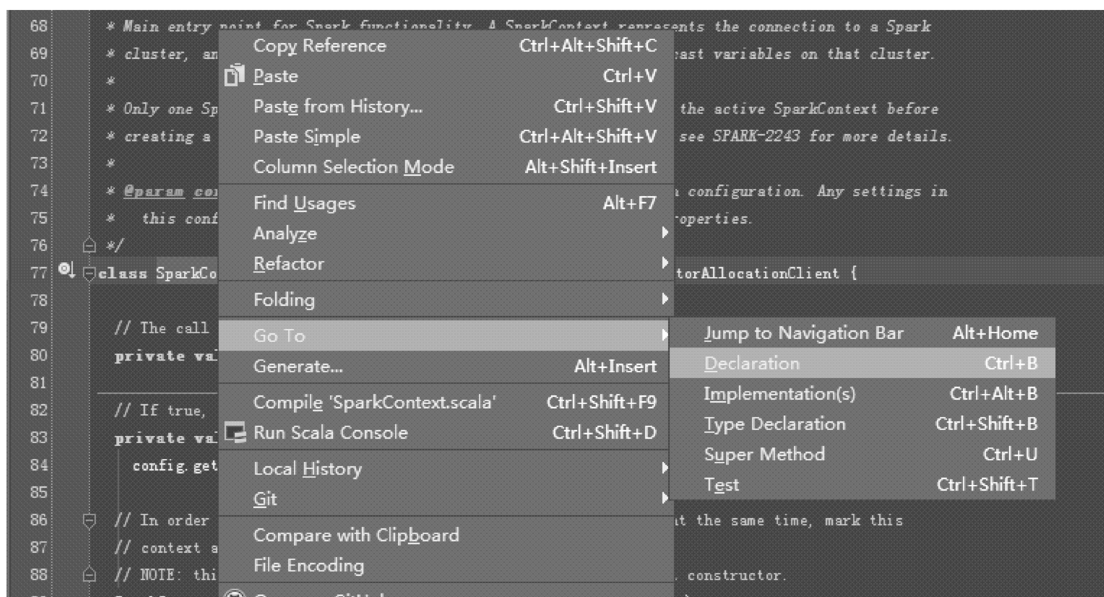


图 2.16 IDEA 中符号声明的调整

通常在 IDE 中阅读源码时，可以通过查看菜单、右键的上下文菜单等信息，来获取比较常用功能的快捷键。

理解 RDD 的 Lineage 关系的案例与源码解析的内容：解析 RDD 中与 Lineage 关系相关的三个组成部分。RDD 抽象概念对 Lineage 关系的天然支持，是构建 Spark DAG 这一高层调度机制的基石。参考章节 1.3RDD 的编程模型，从分区列表、计算每个分片的函数以及对父 RDD 的依赖列表这三个 RDD 的组成部分对 Lineage 关系进行解析：

1） compute：计算每个分片的函数。

2） getPartitions：分区列表。

3） getDependencies：对父 RDD 的依赖列表。

Lineage 关系，即血统关系，可以从两个角度进行解析，一个是构建一个 RDD 的数据流的血统关系，一个是 RDD 构建的算子流的血统关系。

SparkContext 是 Spark 功能的主入口点，命名为 sc，在接下来的案例解析中直接使用 sc 来表示。

一、TextFile 案例与解析

案例代码：

```
scala > val textFile = sc. textFile( "README. md" )
textFile :org. apache. spark. rdd. RDD[ String] = README. md MapPartitionsRDD[ 21 ] at textFile at <
console > :23
scala > textFile. toDebugString
15/04/03 11:59:15 INFOmapred. FileInputFormat:Total input paths to process:1
res49 :String =
(2) README. mdMapPartitionsRDD[ 21 ] at textFile at < console > :23 [ ]
| README. mdHadoopRDD[ 20 ] at textFile at < console > :23 [ ]
scala > textFile. dependencies
```

```
res51: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.OneToOneDependency@8ee9b7)
scala> sc.defaultParallelism
res52: Int = 4
scala> textFile.partitions.size
res53: Int = 2
scala> sc.getConf.getInt("spark.default.parallism",0)
res54: Int = 0
```

代码解析如下：

通过 SparkContext 的 textFile 接口，从外部存储系统加载数据，构建 RDD，即 textFile。调用 toDebugString 查看其 Lineage 的关系：从 HadoopRDD 转换为 MapPartitionsRDD。

获取 RDD 的分区数的方法有两种，获取 Spark 的所有 RDD 的默认分区数的方法 sc.defaultParallelism 和获取 Spark 的具体 RDD 实例的分区数的方法 rdd.partitions.size，如 textFile.partitions.size，这里的 textFile 为加载文件数据后构建的 RDD 实例。

这里使用 sc.getConf.getInt（"spark.default.parallism"，0）来获取当前设置的默认分区数对应的配置属性值。

二、HadoopRDD、MapPartitionsRDD 源码解析

根据 Lineage 的关系从 HadoopRDD 转换为 MapPartitionsRDD，解析源码：

从主入口点 SparkContext 的源码开始解析，打开 IDEA，使用【Ctrl + N】组合键，在弹出窗口中输入 SparkContext，按【Enter】键进入 SparkContext 源码后，查看 textFile 源码，如图 2.17 所示。

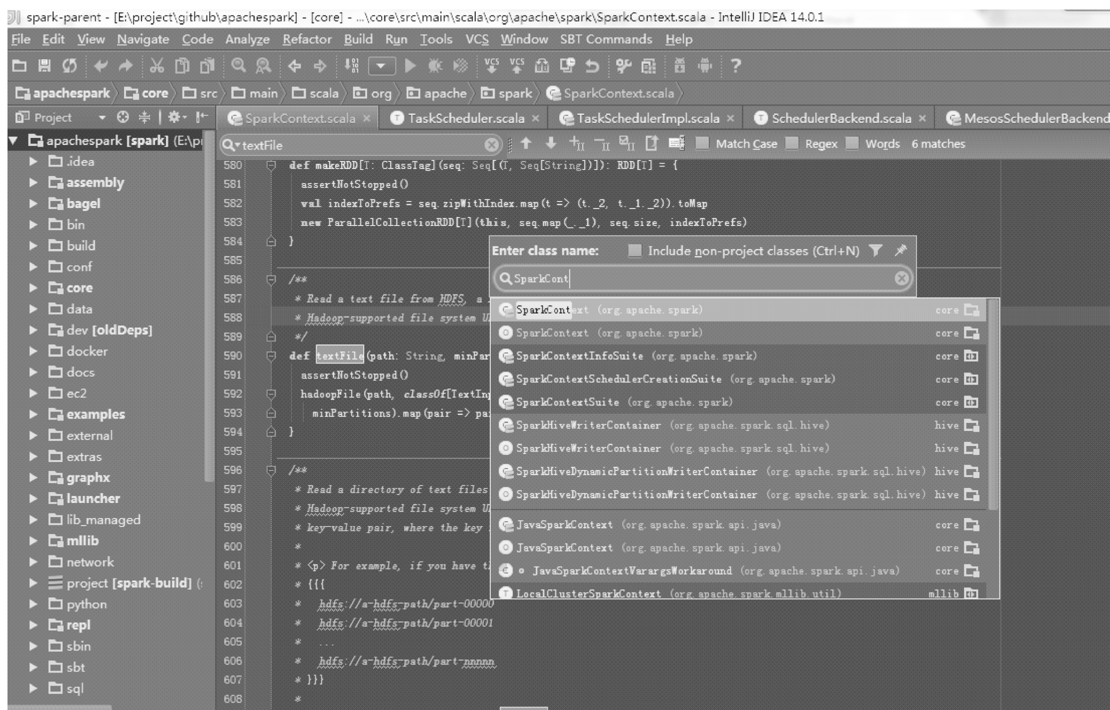


图 2.17　IDEA 中查找、打开类文件

使用【Ctrl + F】组合键，在 SparkContext 源码中查找 textFile 方法，找到的 textFile 方法的源码如下所示：

```
/**
 * Read a text file from HDFS,a local file system (available on all nodes),or any
 * Hadoop-supported file system URI,and return it as an RDD of Strings.
 */
def textFile(path:String,minPartitions:Int = defaultMinPartitions):RDD[String] = {
assertNotStopped()
hadoopFile(path,classOf[TextInputFormat],classOf[LongWritable],classOf[Text],
minPartitions).map(pair => pair._2.toString).setName(path)
}
```

根据 Lineage 的关系：从 HadoopRDD 转换为 MapPartitionsRDD，首先解析 HadoopRDD，按住【Ctrl】键，单击 HadoopRDD 类名，进入 HadoopRDD 的源码，进入的源码如下所示：

```
/** Get an RDD for aHadoop file with an arbitrary InputFormat
 *
 *''' Note:''' Because Hadoop's RecordReader class re-uses the same Writable object for each
 * record,directly caching the returned RDD or directly passing it to an aggregation or shuffle
 * operation will create many references to the same object.
 * If you plan to directly cache,sort,or aggregate Hadoop writable objects,you should first
 * copy them using a'map' function.
 */
def hadoopFile[K,V](
    path:String,
inputFormatClass:Class[_ <:InputFormat[K,V]],
keyClass:Class[K],
valueClass:Class[V],
minPartitions:Int = defaultMinPartitions
    ):RDD[(K,V)] = {
assertNotStopped()
  // A Hadoop configuration can be about 10 KB,which is pretty big,so broadcast it.
  val confBroadcast = broadcast(new SerializableWritable(hadoopConfiguration))
  val setInputPathsFunc = (jobConf:JobConf) => FileInputFormat.setInputPaths(jobConf,path)
  new HadoopRDD(
    this,
confBroadcast,
    Some(setInputPathsFunc),
inputFormatClass,
keyClass,
valueClass,
minPartitions).setName(path)
}
```

继续查看 HadoopRDD 类，按住【Ctrl】键，单击 HadoopRDD 类名，进入 HadoopRDD 的源码，如下所示，在 HadoopRDD 的源码中可以看到，对应 HadoopRDD 的分区类为 HadoopPartition（根据类的名字，以及 getPartitions 方法中构建的分区的类型），这是 getPartitions 方法中需要构建的分区类型：

```
@ DeveloperApi
class HadoopRDD[K,V](
    sc:SparkContext,
broadcastedConf:Broadcast[SerializableWritable[Configuration]],
    initLocalJobConfFuncOpt:Option[JobConf => Unit],
inputFormatClass:Class[_ <:InputFormat[K,V]],
keyClass:Class[K],
valueClass:Class[V],
minPartitions:Int)
  extends RDD[(K,V)](sc,Nil) with Logging {
    ……
```



通过 deps 成员查看 HadoopRDD 的父依赖：在 HadoopRDD 的主构造函数中，我们可以看到其依赖的父 RDD（即类的 deps 成员）为 Nil，即 HadoopRDD 的父依赖 RDD 为空，这是因为它是从外部数据加载的，而不是从其他 RDD 转换得到的。

继续查看重载的 getPartitions 方法，解析 HadoopRDD 是如何记录各个分区的数据来源信息：

```
override defgetPartitions:Array[Partition] = {
  val jobConf = getJobConf()
  // add the credentials here as this can be called before SparkContext initialized
SparkHadoopUtil.get.addCredentials(jobConf)
  val inputFormat = getInputFormat(jobConf)
  if (inputFormat.isInstanceOf[Configurable]) {
inputFormat.asInstanceOf[Configurable].setConf(jobConf)
  }
  val inputSplits = inputFormat.getSplits(jobConf,minPartitions)
  val array = new Array[Partition](inputSplits.size)
  for (i <- 0 until inputSplits.size) {
    array(i) = new HadoopPartition(id,i,inputSplits(i))
  }
  array
}
```

其中，构建 HadoopRDD 分区时传入的 minPartitions 参数（参数表示最小分区个数）会作为输入文件 split 的个数的参考值。可以看到在代码中使用 inputSplits 构建出（即 new 出 HadoopRDD 的实例）HadoopRDD 的全部分区。

到这一步，可以得出 HadoopRDD 的分区 HadoopParition 与 Hadoop 的 InputFormat 的 Split 一一对应，即 HadoopRDD 各个分区的数据来源对应于 InputFormat 指定的 Split，通常对应于 Hadoop 文件的分块。

继续解析 Hadoop 的 compute 方法，该方法指定对分区的数据源的读取方式，并没有真正开始读取，在 Action 触发之后，调用该 compute 方法获取 Iterator 时才会开始读取数据。

```
override def compute(theSplit:Partition,context:TaskContext):InterruptibleIterator[(K,V)] = {
  val iter = new NextIterator[(K,V)] {
    val split = theSplit.asInstanceOf[HadoopPartition]
logInfo("Input split:" + split.inputSplit)
```

```
    val jobConf = getJobConf()
    val inputMetrics = context.taskMetrics
      .getInputMetricsForReadMethod(DataReadMethod.Hadoop)
    // Find a function that will return the FileSystem bytes read by this thread. Do this before
    // creating RecordReader,because RecordReader' s constructor might read some bytes
    val bytesReadCallback = inputMetrics.bytesReadCallback.orElse {
      split.inputSplit.value match {
        case _:FileSplit | _:CombineFileSplit =>
SparkHadoopUtil.get.getFSBytesReadOnThreadCallback()
        case _ => None
      }
    }
inputMetrics.setBytesReadCallback(bytesReadCallback)
    var reader:RecordReader[K,V] = null
    val inputFormat = getInputFormat(jobConf)
HadoopRDD.addLocalConfiguration(new SimpleDateFormat("yyyyMMddHHmm").format(create-
Time),
      context.stageId,theSplit.index,context.attemptNumber,jobConf)
    reader = inputFormat.getRecordReader(split.inputSplit.value,jobConf,Reporter.NULL)
    // Register an on-task-completion callback to close the input stream.
    context.addTaskCompletionListener{ context => closeIfNeeded() }
    val key:K = reader.createKey()
    val value:V = reader.createValue()
    override def getNext() = {
      try {
        finished = ! reader.next(key,value)
      } catch {
        case eof:EOFException =>
          finished = true
      }
      if (! finished) {
inputMetrics.incRecordsRead(1)
      }
      (key,value)
    }
    override def close() {
      try {
        reader.close()
        if (bytesReadCallback.isDefined) {
inputMetrics.updateBytesRead()
        } else if (split.inputSplit.value.isInstanceOf[FileSplit] ||
                   split.inputSplit.value.isInstanceOf[CombineFileSplit]) {
          // If we can' t get the bytes read from the FS stats,fall back to the split size,
          // which may be inaccurate.
          try {
inputMetrics.incBytesRead(split.inputSplit.value.getLength)
          } catch {
            case e:java.io.IOException =>
logWarning("Unable to get input size to set InputMetrics for task",e)
```



```
                }
              }
            } catch {
              case e:Exception => {
                if (! Utils. inShutdown( ) ) {
logWarning( "Exception in RecordReader. close( )" ,e)
                }
              }
            }
          }
        }
      new InterruptibleIterator[ (K,V) ]( context,iter)
    }
```

在代码中，构建分区数据的 Iterator 时，使用 RecordReader 来读取当前 Split 的数据。至此，HadoopRDD 的分区构建和分区计算都已经解析完成。继续 MapPartitionsRDD 源码的解析，对应 MapPartitionsRDD 源码的入口点如下：

```
/ **
  * Read a text file from HDFS,a local file system (available on all nodes),or any
  * Hadoop - supported file system URI,and return it as an RDD of Strings.
  */
def textFile( path:String,minPartitions:Int = defaultMinPartitions) :RDD[ String] = {
assertNotStopped( )
hadoopFile( path,classOf[ TextInputFormat],classOf[ LongWritable],classOf[ Text],
minPartitions). map( pair => pair. _2. toString). setName( path)
}
```

在 textFile 方法中通过 map 方法将 HadoopRDD 再次转换为 MapPartitionsRDD。查看 map 方法的源码，可以看到 HadoopRDD 被转换为 MapPartitionsRDD：

```
/ **
  * Return a new RDD by applying a function to all elements of this RDD.
  */
def map[ U:ClassTag]( f:T => U) :RDD[ U] = {
  val cleanF = sc. clean( f)
  new MapPartitionsRDD[ U,T]( this,( context,pid,iter) => iter. map( cleanF) )
}
```

其中，传入的参数 f：T => U 被作用到 iter 上。

按【Ctrl】键，单击 MapPartitionsRDD，进入 MapPartitionsRDD 源码：

```
private[ spark] classMapPartitionsRDD[ U:ClassTag,T:ClassTag](
    prev:RDD[ T],
    f:( TaskContext,Int,Iterator[ T]) => Iterator[ U],   // ( TaskContext,partition index,iterator)
    preservesPartitioning:Boolean = false)
  extends RDD[ U]( prev) {
  override val partitioner = if (preservesPartitioning) firstParent[ T]. partitioner else None
  override def getPartitions:Array[ Partition] = firstParent[ T]. partitions
  override def compute( split:Partition,context:TaskContext) =
```

```
        f(context,split.index,firstParent[T].iterator(split,context))
    }
```

同样，通过 MapPartitionsRDD 主构造函数可以得到依赖的父 RDD，当前传入的 RDD 为 HadoopRDD。

查看分区方法 getPartitions 源码，解析各个分区的数据来源：firstParent 为传入的父 RDD，getPartitions 记录了 MapPartitionsRDD 分区的数据从依赖的父 RDD 分区中获取的关系。

查看重载的 compute 方法：记录对分区数据来源的操作，MapPartitionsRDD 对父 RDD 分区的执行传入的 f 操作。

通过以上的解析，可以得到从外部存储系统的数据集到 MapPartitionsRDD 的整个 Lineage 的关系如图 2.18 所示。

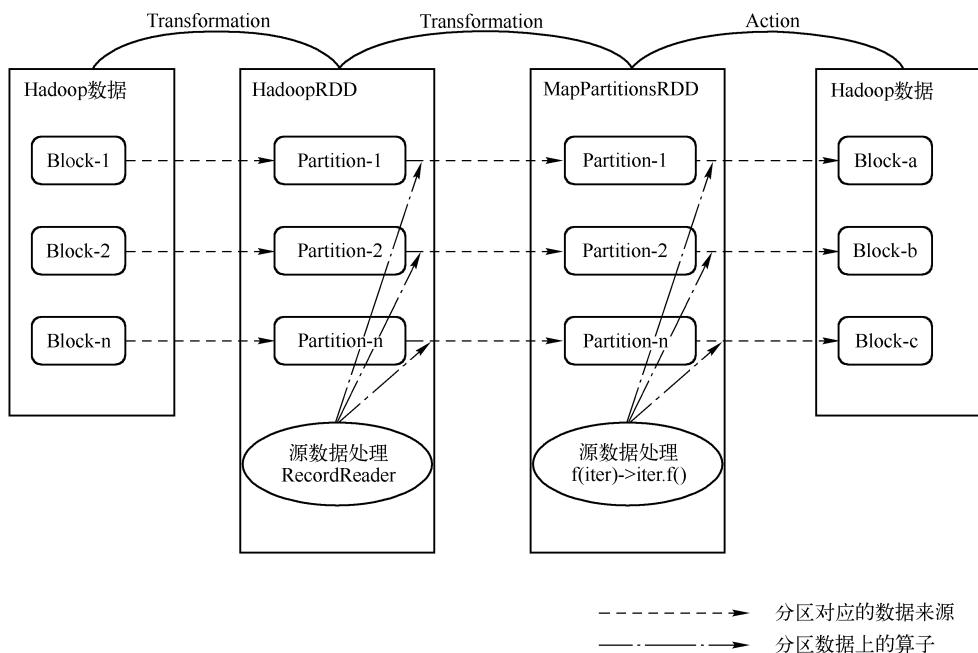


图 2.18　Lineage 数据流与操作算子流图

上图对应了前面的构建出 MapPartitionsRDD 的整个 Lineage 关系的过程，从该图中可以解析出通用的 Lineage 关系，具体从 Lineage 的数据流和处理流两个角度进行解析，即 RDD 的血统关系，记录了构建 RDD 时，各个点上的数据的流动关系，以及数据从上一个点到下一个点流动时，经过了什么样的处理，对应的两个关系图如下：

1）数据流关系图：每一个 RDD 记录了内部各个分区的数据来源，数据来源可以是外部存储系统，也可以是其他依赖的父 RDDs，通过上图中的分区对应的数据来源的虚线箭头，可以看到 Block 数据和 Partition 数据之间的数据流关系。

2）处理流关系图：当该 RDD 作为其他 RDD 的数据来源，或作为外部存储系统的数据来源时，该 RDD 对各个分区数据的处理，通过上图中的分区数据上的算子的虚线箭头，可以看到 Block 数据和 Partition 数据之间在数据流动时，在流动数据上进行的算子操作，即对这些流动数据的处理。

整个 Lineage 记录了数据流关系和处理流关系，仅仅在 Action 操作时，才会将记录的关系提交到 Job 上，真正执行起来。

三、ReduceByKey 的案例与源码解析

MapPartitionsRDD 是一个窄依赖，为了进一步解析 Lineage 机制中的宽依赖，基于 reduceByKey 转换操作来解析如何形成 ShuffledRDD。

查看 reduceByKey 的 lineage 关系图：

```
scala > wordCounts. toDebugString
res4:String =
(1)ShuffledRDD[4] at reduceByKey at <console>:24 [Memory Deserialized 1x Replicated]
 + -(1)MapPartitionsRDD[3] at map at <console>:24 [Memory Deserialized 1x Replicated]
    |MapPartitionsRDD[2] at flatMap at <console>:24 [Memory Deserialized 1x Replicated]
    |  /README. mdMapPartitionsRDD[1] at textFile at <console>:21 [Memory Deserialized 1x
Replicated]
    |  /README. mdHadoopRDD[0] at textFile at <console>:21 [Memory Deserialized 1x Replica-
ted]
```

其中，[Memory Deserialized 1x Replicated]是该 RDD 的缓存级别信息，在调用了 cache 等持久化操作后就会出现该信息。

MapPartitionsRDD 已经解析过，下面解析 ShuffledRDD。ShuffledRDD 源码如下：

```
@DeveloperApi
class ShuffledRDD[K,V,C](
    @transient var prev:RDD[_ <:Product2[K,V]],
    part:Partitioner)
  extends RDD[(K,C)](prev. context,Nil) {
  private var serializer:Option[Serializer] = None
  private var keyOrdering:Option[Ordering[K]] = None
  private var aggregator:Option[Aggregator[K,V,C]] = None
  private var mapSideCombine:Boolean = false
    ……
```

从源码可知，ShuffledRDD 的分区类型为 ShuffledRDDPartition。从 ShuffledRDD 的主构造函数可知，其父依赖为传入的 MapPartitionsRDD。

对应 RDD 的三个主要方法，源码如下：

```
override defgetDependencies:Seq[Dependency[_]] = {
  List(new ShuffleDependency(prev,part,serializer,keyOrdering,aggregator,mapSideCombine))
}
override val partitioner = Some(part)
override def getPartitions:Array[Partition] = {
  Array. tabulate[Partition](part. numPartitions)(i => new ShuffledRDDPartition(i))
}
override def compute(split:Partition,context:TaskContext):Iterator[(K,C)] = {
  val dep = dependencies. head. asInstanceOf[ShuffleDependency[K,V,C]]
SparkEnv. get. shuffleManager. getReader(dep. shuffleHandle,split. index,split. index + 1,context)
     . read()
     . asInstanceOf[Iterator[(K,C)]]
}
```

在 getDependencies 方法中，将父依赖 RDD 封装到了 ShuffleDependency 中，在 DAG 调度时，DAG 是根据 Shuffle 来划分 Stage 的，而是否为 Shuffle，则是通过判断 RDD 对父依赖的依赖类型是否为 ShuffleDependency。

查看 ShuffleDependency 的源码：

```
@DeveloperApi
class ShuffleDependency[K,V,C](
    @transient _rdd:RDD[_ <:Product2[K,V]],
    val partitioner:Partitioner,
    val serializer:Option[Serializer] = None,
    val keyOrdering:Option[Ordering[K]] = None,
    val aggregator:Option[Aggregator[K,V,C]] = None,
    val mapSideCombine:Boolean = false)
  extends Dependency[Product2[K,V]] {
  override def rdd = _rdd.asInstanceOf[RDD[Product2[K,V]]]
  val shuffleId:Int = _rdd.context.newShuffleId()
  val shuffleHandle:ShuffleHandle = _rdd.context.env.shuffleManager.registerShuffle(
shuffleId,_rdd.partitions.size,this)
  _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}
```

可以看到，每个 ShuffleDependency 分配了 SparkContext 中全局唯一的 ID 值 shuffledId，可以通过这唯一的标识来找到 Shuffle 数据；同时，在构造 ShuffleDependency 时，向 ShuffleManager 注册了一个 ShuffleHandle，它负责计算过程中，Shuffle 过程的读写操作。

查看 gePartitions 方法的源码，可以看到该方法只是根据分区器 part，构建了 ShuffledRDD 的各个分区，记录了数据来源；查看 compute 方法，该方法中使用了之前在 ShuffleManager 注册过的 ShuffleHandle，用 ShuffleHandle 的 Reader 来实际获取数据。

这部分可以参考下之前对 HadoopRDD 的 compute（方法名），都是使用某种 Reader（读取数据的类名）来获取数据，并没有对数据做处理。

下面通过 DAG、ShuffledRDD 的简单执行图，来加深解析。执行流程如图 2.19 所示。

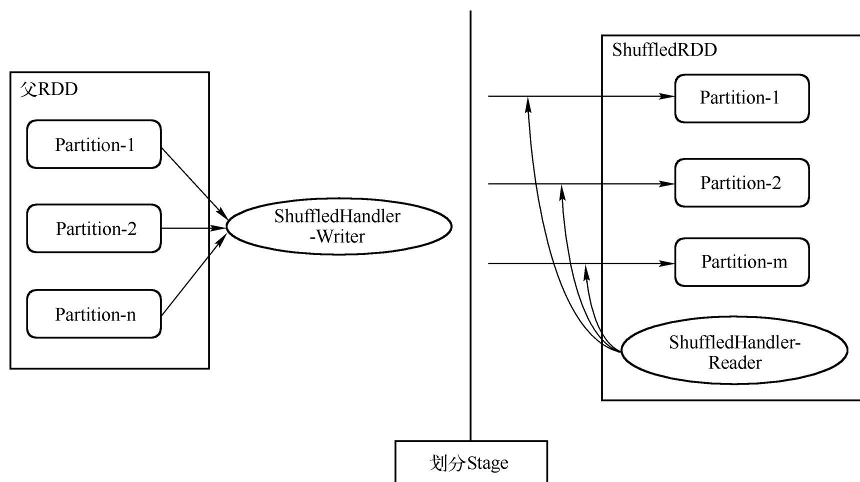


图 2.19　Stage 划分以及父子 RDD 数据流图

由上图可以看到，当 DAG 碰到 Shuffle 依赖时，会切分 Stage，Stage 前的 RDD 分区数据会通过 ShuffledDependency 的 shuffleId 作为 Shuffle 的唯一标识，利用 ShuffleHander 的 Writer（写数据的类名）写数据。Stage 的 RDD 在读取各个分区数据时，compute 方法使用 ShuffleHander 的 Reader，根据 shuffleId 去读取数据。

2.2.7 RDD 的持久化案例与解析

可以使用 persist 方法和 cache 方法将任意 RDD 缓存到内存或磁盘、Tachyon 文件系统中；其中，cache 方法是 persist 方法使用 MEMORY_ONLY 存储级别的快捷方式。

RDD 对应的缓存是容错的，如果一个 RDD 分片丢失，可以通过构建它的 Transformation 自动重构。

这一节详细描述了可用的存储级别及其含义，并对使用默认存储级别的 cache 方法进行详细解析；同时，以 MEMORY_AND_DISK 存储级别为例，扩展 persist 方法的应用及不同缓存特点，包括缓存的 lazy 特性、内存无法缓存时的处理方法等，解析过程中对比了 MEMORY_ONLY 与 MEMORY_AND_DISK 这两个等级的缓存结果的差异点。

具体操作步骤如下：

一、小文件缓存解析

这里的小文件的是指该文件的数据大小在当前内存中可以全部装载。

1. 加载文件，输入命令：

```
vallinesWithSpark = textFile. filter( line => line. contains(“Spark”) )
```

2. 缓存 RDD 到内存中，输入命令：

```
linesWithSpark. cache( )
```

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，真正缓存之前的内存信息界面如图 2.20 所示。

Spark 1.3.0　Jobs　Stages　Storage　Environment　Executors　Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk

图 2.20　Driver Program 中缓存 RDD 之前的 Storage 界面

可以看到执行 cache 后，并不会马上进行缓存。

3. 用 Action 操作触发缓存，输入 count 命令，触发前面的 cache 的执行。

```
scala > linesWithSpark. count( )
15/04/25 11:18:30 INFOSparkContext:Starting job:count at <console>:27
15/04/25 11:18:30 INFODAGScheduler:Got job 5 (count at <console>:27) with 1 output partitions
(allowLocal = false)
15/04/25 11:18:30 INFODAGScheduler:Final stage:Stage 5(count at <console>:27)
15/04/25 11:18:30 INFODAGScheduler:Parents of final stage:List( )
15/04/25 11:18:30 INFODAGScheduler:Missing parents:List( )
```

```
15/04/25 11:18:30 INFODAGScheduler:Submitting Stage 5 (MapPartitionsRDD[5] at filter at <console>:24),which has no missing parents
15/04/25 11:18:30 INFOSparkContext:Created broadcast 6 from broadcast at DAGScheduler.scala:839
15/04/25 11:18:30 INFODAGScheduler:Submitting 1 missing tasks from Stage 5 (MapPartitionsRDD[5] at filter at <console>:24)
15/04/25 11:18:30 INFOTaskSchedulerImpl:Adding task set 5.0 with 1 tasks
15/04/25 11:18:30 INFOTaskSetManager:Starting task 0.0 in stage 5.0 (TID 5,localhost,ANY,1295 bytes)
15/04/25 11:18:30 INFO Executor:Running task 0.0 in stage 5.0 (TID 5)
15/04/25 11:18:30 INFOCacheManager:Partition rdd_5_0 not found,computing it
15/04/25 11:18:30 INFOHadoopRDD:Input split:hdfs://cluster04:9000/README.md:0+3629
15/04/25 11:18:30 INFO Executor:Finished task 0.0 in stage 5.0 (TID 5). 2399 bytes result sent to driver
15/04/25 11:18:30 INFODAGScheduler:Stage 5 (count at <console>:27) finished in 0.022s
15/04/25 11:18:30 INFODAGScheduler:Job 5 finished:count at <console>:27,took 0.032341s
res16:Long = 19
```

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，触发并缓存之后的内存信息如图 2.21 所示。

Spark 1.3.0 Jobs Stages Storage Environment Executors — Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
2	Memory Deserialized 1x Replicated	2	100%	3.3 KB	0.0 B	0.0 B

图 2.21　Driver Program 中缓存 RDD 后的 Storage 界面

可以看到，全部分区都已经缓存到 Memory 中。

4. 执行 unpersist 方法，释放缓存。

```
linesWithSpark.unpersist
```

执行过程如下：

```
scala>linesWithSpark.unpersist()
15/04/25 11:19:39 INFOMapPartitionsRDD:Removing RDD 5 from persistence list
res17:linesWithSpark.type = MapPartitionsRDD[5] at filter at <console>:24
```

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，eager 型的 unpersist 操作后的内存信息如图 2.22 所示。

Spark 1.3.0 Jobs Stages Storage Environment Executors — Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk

图 2.22　Driver Program 中 unpersistrdd 后的 Storage 界面

不同于 cache 的 lazy 特性，unpersist 是个 eager 操作，执行后，Web Interface 界面上显示的内存马上被释放了。

二、大文件缓存解析

测试的大数据量的文件缓存到默认存储级别（Memory）。当文件数据量比较小时，全部缓存到了内存中。当大数据量的文件缓存时，在内存不能完全装载的情况下，部分分区数据会被丢弃，可以通过下面步骤进行验证。

1. 查看文件大小，文件有918MB，使用 du -m 目标文件来查看。

```
$ su -m /xdr_wuhan_2020150311.csv
918./xdr_wuhan_2020150311.csv
```

2. 重新加载大数据量的文件，查看 RDD 分区数。

```
scala > val bigText = sc.textFile("/home/harli/cluster_13_241/data/xdr_wuhan_2020150311.csv")
bigText: org.apache.spark.rdd.RDD[String] =/home/harli/cluster_13_241/data/xdr_wuhan_2020150311.csv MapPartitionsRDD[11] at textFile at <console>:21

scala > bigText.partitions.size
15/03/25 15:3:57 INFO FileInputFormat:Total inputpaths to process:1
res32:Int = 29
```

可以看到，bigText. partitions. size 返回 29，也就是当前分区数为 29。

3. 使用默认存储级别进行 cache。

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，使用默认存储级别，在真正缓存之前的内存信息如图 2. 23 所示。

```
scala > bigText.cache
res33:bigText.type =/home/harli/cluster_13_241/data/xdr_wuhan_2020150311.csv MapPartitionsRDD[11] at textFile at <console>:21
```

Spark 1.3.0 | Jobs | Stages | Storage | Environment | Executors | Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
2	Memory Deserialized 1x Replicated	2	100%	3.3 KB	0.0 B	0.0 B
CHANGES.txt	Memory Deserialized 1x Replicated	2	100%	758.4 KB	0.0 B	0.0 B

图 2. 23 Driver Program 中缓存 rdd 前的 Storage 界面

4. 用 take 的 Action 操作触发缓存，语句如下所示。

```
bigText.take(1)
```

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，使用默认存储级别，在真正缓存之后的内存信息如图 2. 24 所示。其中 driverhost 为应用程序启动的节点。

此时，缓存分区数为 1、分区总数为 29、缺失分区数为 28。其中，缺失的分区数在需要使用时会重新计算。

5. 用 count 的 Action 操作触发缓存，语句如下所示。

```
bigText.count
```

查看输出日志信息，由于当前文件的大小超过了内存可以装载的最大值，在缓存过程

Spark 1.3.0 Jobs Stages Storage Environment Executors Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
2	Memory Deserialized 1x Replicated	2	100%	3.3 KB	0.0 B	0.0 B
CHANGES.txt	Memory Deserialized 1x Replicated	2	100%	758.4 KB	0.0 B	0.0 B
/home/harli/cluster_13_241/data/xdr_wuhan_2020150311.csv	Memory Deserialized 1x Replicated	1	3%	69.5 MB	0.0 B	0.0 B

图 2. 24 Driver Program 中 take 方式触发缓存 RDD 后的 Storage 界面

中，我们会在日志中看到内存空间不足的警告日志，警告日志信息会包含“Not enough space to cache rdd_ ... in memory…”这些内容。即，当可用的缓存空间不足以装载 RDD 的分区数据时，就会出现该日志信息，对应的，此时 RDD 可能仅仅缓存了部分的分区，如果指定仅在内存中缓冲的话，其他分区将不会缓存，在下次使用时会重新计算。

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，内存不足以装载整个 RDD 时，仅缓存部分分区数据时的内存信息如图 2. 25 所示。

Spark 1.3.0 Jobs Stages Storage Environment Executors Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
/home/harli/cluster_13_241/data/xdr_wuhan_2020150311.csv	Memory Deserialized 1x Replicated	3	10%	180.9 MB	0.0 B	0.0 B
2	Memory Deserialized 1x Replicated	2	100%	3.3 KB	0.0 B	0.0 B
CHANGES.txt	Memory Deserialized 1x Replicated	2	100%	758.4 KB	0.0 B	0.0 B

图 2. 25 Driver Program 中部分缓存 RDD 后的 Storage 界面

三、修改存储级别，测试不同等级的处理

Spark 提供的存储级别可以参考官方网站中的编程指南部分，具体内容如表 2. 3 所示。

表 2. 4 Spark 提供的存储级别及其具体含义

存储级别（Storage Level）	含　义
MEMORY_ONLY	将 RDD 以反序列化（deserialized）的 Java 对象存储到 JVM。如果 RDD 不能被内存装下，一些分区将不会被缓存，并且在需要的时候被重新计算。这是默认的级别
MEMORY_AND_DISK	将 RDD 以反序列化的 Java 对象存储到 JVM。如果 RDD 不能被内存装下，超出的分区将被保存在硬盘上，并且在需要时被读取
MEMORY_ONLY_SER	将 RDD 以序列化（serialized）的 Java 对象进行存储（每一分区占用一个字节数组）。通常来说，这比将对象反序列化的空间利用率更高，尤其当使用快速序列化器（fast serializer），但在读取时会比较耗 CPU
MEMORY_AND_DISK_SER	类似于 MEMORY_ONLY_SER，但是把超出内存的分区将存储在硬盘上而不是在每次需要的时候重新计算
DISK_ONLY	只将 RDD 分区存储在硬盘上
MEMORY_ONLY_2 MEMORY_AND_DISK_2	与上述的存储级别一样，但是将每一个分区都复制到两个集群节点上
OFF_HEAP（experimental）	以序列化的格式将 RDD 存储到 Tachyon。相比于 MEMORY_ONLY_SER，OFF_HEAP 降低了垃圾收集（Garbage Collection）的开销，并使 Executors 变得更小而且共享内存池，这在大堆（heaps）和多应用并行的环境下是非常吸引人的。而且，由于 RDD 驻留于 Tachyon 中，Executor 的崩溃不会导致内存中的缓存丢失。在这种模式下，Tachyon 中的内存是可丢弃的。因此，Tachyon 不会尝试重建一个在内存中被清除的分块

以 MEMORY_AND_DISK 进行测试，具体步骤如下。

1）首先导入存储级别类：

```
import org. apache. spark. storage. StorageLevel. _
```

2）以 MEMORY_AND_DISK 级别缓存 RDD，输入命令：

```
bigText. persist(MEMORY_AND_DISK)
```

3）Action 触发缓存，输入命令：

```
bigText. take(1)
```

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，在 Action 触发后真正缓存 RDD 时的内存信息如图 2. 26 所示。

Spark 1.3.0 Jobs Stages Storage Environment Executors Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
/home/harli/cluster_13_241/data/xdr_wuhan_2020150311.csv	Memory Deserialized 1x Replicated	1	3%	69.5 MB	0.0 B	0.0 B
2	Memory Deserialized 1x Replicated	2	100%	3.3 KB	0.0 B	0.0 B
CHANGES.txt	Memory Deserialized 1x Replicated	2	100%	758.4 KB	0.0 B	0.0 B

图 2. 26　Driver Program 中 take 方式触发缓存 RDD 后的 Storage 界面

可以看到，当前缓存的分区数为 1，其他未缓存的分区在计算时需重新计算。

解析：cache 是一个标志性操作，仅在数据真正计算时，才会缓存到内存，因此即使存储级别为 MEMORY_AND_DISK，但 take 操作只需要针对一个分区进行操作，对应的，Storage 也仅缓存了一个分区。

4）count 操作进行 Action 触发缓存，输入命令：

```
bigText. count
```

查看 Web Interface 界面（http://driverhost:4040）的 Storage 部分，count 方式触发 RDD 缓存时的内存信息如图 2. 27 所示。

vel	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
ed 1x Replicated	29	100%	177.3 MB	0.0 B	838.8 MB
erialized 1x Replicated	2	100%	3.3 KB	0.0 B	0.0 B
erialized 1x Replicated	2	100%	758.4 KB	0.0 B	0.0 B

图 2. 27　Driver Program 中 count 方式触发缓存 RDD 后的 Storage 界面

可以看到缓存的分区数为 29，其中一部分缓存在 Memory，一部分缓存在 Disk，其他未缓存的分区在计算时需重新计算。

解析：与 take 类似，由于 count 操作涉及各个分区数据的计算，因此最终会导致全部分区都处理之前的 cache 标志信息，即，存储在内存和磁盘中。

在 RDD Name 列下单击具体的 RDD Name，可以进入各个分区数据缓存的具体信息，部

分分区数据会存储在内存中，内存装载不下的分区数据会缓存到磁盘中。

2.2.8 RDD 的构建案例与解析

一、加载外部存储系统的文件构建 RDD

首先，textFile 方法构建时，支持多种方式，包括目录加载、通配符加载等，如：

1） textFile （"/my/directory"）：加载指定目录下的所有文件。

2. textFile （"/my/directory/ * . txt"）：加载指定目录下所有 txt 格式的文件。

3） textFile （"/my/directory/ * . gz"）：下载指定目录下所有 . gz 格式的文件。

其次，Spark 基于 HDFS 存储系统时，可以加载任何支持 HadoopInputFormat 格式的文件输入，同时也支持任何以 HadoopOutputFormat 格式的文件输出。下面以加载时不同的 HadoopInputFormat 来说明 Spark 对不同文件格式的支持。类似的，当 Spark 输出到外部存储系统时，对应不同格式应使用相应的 HadoopOutputFormat 子类。

小技巧：文件路径中，目录的通配符使用“ ** ”，比如 val file = "hdfs://wh001:8020//user/harli/ ** / * . * "，对应于 hdfs://wh001:8020//user/harli/下所有子目录下的所有“ * . * ”格式的文件。

下面举例分析不同文件格式对应的 HadoopInputFormat 子类设置。

比如 textFile 和 sequenceFile 这两个方法，其底层都是调用了 hadoopFile 接口，只是使用的加载参数不同，下面从源码上进行分析。

textFile 的源码：

```
/**
 * Read a text file from HDFS,a local file system (available on all nodes),or any
 * Hadoop - supported file system URI,and return it as an RDD of Strings.
 */
def textFile(path:String,minPartitions:Int = defaultMinPartitions):RDD[String] = {
assertNotStopped()
hadoopFile(path,classOf[TextInputFormat],classOf[LongWritable],classOf[Text],
minPartitions). map(pair => pair. _2. toString). setName(path)
}
```

textFile 方法调用 hadoopFile 时，对应的 InputFormat 设置为 TextInputFormat，即文本文件的输入。

sequenceFile 的源码：

```
defsequenceFile[K,V](path:String,
keyClass:Class[K],
valueClass:Class[V],
minPartitions:Int
    ):RDD[(K,V)] = {
assertNotStopped()
  val inputFormatClass = classOf[SequenceFileInputFormat[K,V]]
hadoopFile(path,inputFormatClass,keyClass,valueClass,minPartitions)
}
```

sequenceFile 方法调用 hadoopFile 时，对应的 InputFormat 设置为 SequenceFileInputFormat，即序列文件的输入。

最后，关于文件压缩方式，如 textFile（"/my/directory/ * . gz"），由于 . gz 格式的压缩文件是 Hadoop 内置支持的（根据扩展名自动识别），因此可以直接加载。但如果使用不同压缩格式时，应该修改 HDFS 的配置信息，具体配置信息请查询具体版本的 Hadoop 官方文档。

注意： 内置的 . gz 压缩格式不支持 split。

二、从 Scala 数据集构建 RDD

除了使用如案例中的 textFile 方法加载外部存储系统的数据来构建 RDD 外，RDD 还可以从 Scala 数据集构建，下面使用 SparkContext 的 parallelize 方法来构建 RDD，代码如下：

```
scala > valstrRdd = sc. parallelize ( List ("a","b","c","c","e") ,2)
strRdd:org. apache. spark. rdd. RDD[String] = ParallelCollectionRDD[11] at parallelize at <console>:
21
scala > strRdd. partitions. size
res5:Int =2
scala > strRdd. toDebugString
res6:String =(2) ParallelCollectionRDD[11] at parallelize at <console>:21 []
```

Parallelize 的第二个参数可以用于设置构建的 RDD 的分区个数，当该参数未指定时，使用 SparkContext 的默认并行数。当 Parallelize 的分区参数未指定时，具体可见下面的源码分析。

2.2.9 分区数设置的案例与源码解析

解析各种情况下的 RDD 的分区数设置情况，具体包括加载文件创建的 RDD 的分区数、SparkContext 中默认的分区数、经过转换后的 RDD 的分区数以及针对特定的 Key – Value 格式类型的 RDD 的分区数。

一、加载文件创建 RDD 时的分区数解析

如下所示：

```
scala > val textFile = sc. textFile("../README. md")
scala > textFile. toDebugString
15/03/31 02:06:45 INFOFileInputFormat:Total input paths to process:1
res7:String =
(2) ../README. md MapPartitionsRDD[1] at textFile at <console>:21 []
|  ../README. mdHadoopRDD[0] at textFile at <console>:21 []
scala > textFile. dependencies
res8: Seq [ org. apache. spark. Dependency [ _]]  = List ( org. apache. spark. OneToOneDependency @
3911e00f)
scala > textFile. partitions. size
res9:Int =4
scala > sc. defaultParallelism
res10:Int =4
```

```
scala > sc. getConf. getInt( "spark. default. parallelism" ,0)
res11:Int =0
```

sc. defaultParallelism 是当前默认的并行数，对应加载的本地文件（非 HDFS 文件系统）以默认的并行数作为构建的 RDD 的分区数。

注意：当前没有设置“spark. default. parallelism”配置属性，所以使用 sc. getConf. getInt（"spark. default. parallelism"，0）获取属性时，值为设置的默认值 0。

查看 SparkContext 的源码：

```
/** Default level of parallelism to use when not given by user (e. g. parallelize andmakeRDD). */
def defaultParallelism:Int = {
assertNotStopped( )
taskScheduler. defaultParallelism
}
```

跳转到 taskScheduler. defaultParallelism 源码：

```
// Get the default level of parallelism to use in the cluster,as a hint for sizing jobs.
def defaultParallelism( ):Int
```

单击方法前的箭头，获取其具体子类的实现，当前仅提供一个具体子类，即 TaskSchedulerImpl，如图 2. 28 所示。

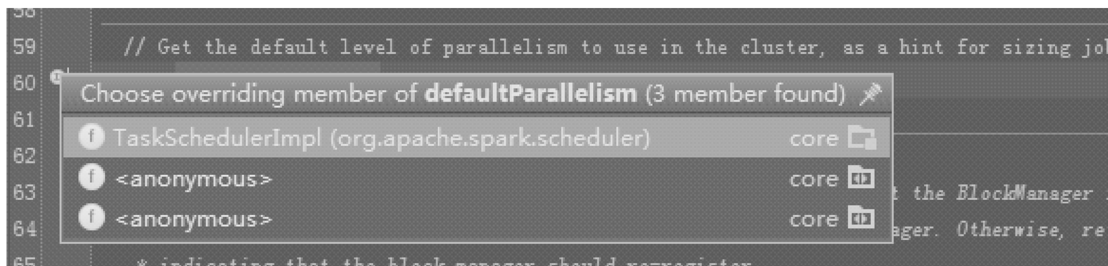


图 2. 28　获取 taskScheduler. defaultParallelism 方法的具体重载信息

跳转到 TaskSchedulerImpl 的源码：

```
override defdefaultParallelism( ):Int = backend. defaultParallelism( )
```

继续跳转到 backend. defaultParallelism，然后跳转到具体子类的实现，继承的全部子类如图 2. 29 所示。

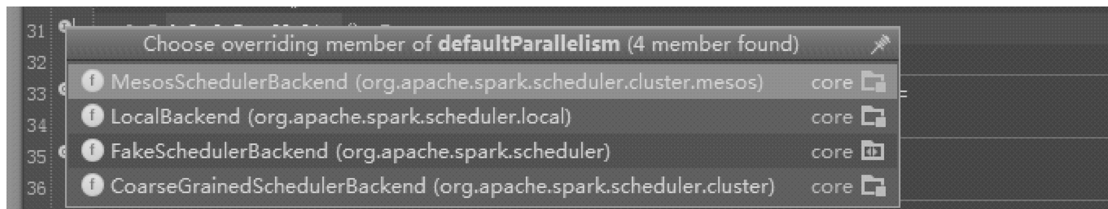


图 2. 29　获取重载 backend. defaultParallelism 方法的具体子类

如 MesosSchedulerBackend（对应集群模式）中的源码所示：

```
// TODO:queryMesos for number of cores
override def defaultParallelism( ):Int = sc. conf. getInt( "spark. default. parallelism" ,8)
```

可以看到 defaultParallelim 的配置属性为："spark. default. parallelism"，即集群模式下使用该属性作为 defaultParallelim，并且默认值为 8。



如子类 LocalBackend 中的源码（对应 Local 模式）所示：

```
override defdefaultParallelism( ):Int =
  scheduler. conf. getInt( "spark. default. parallelism" ,totalCores)
```

可以看到 defaultParallelim 的配置属性为："spark. default. parallelism"，但当没有设置该配置属性时，默认值为总的内核数，LocalBackend 对应 Local 模式。

修改默认的配置文件：$ _HOME/conf/spark - defaults. conf，在文件中添加该属性的配置，如下所示：

```
# Default system properties included when running spark - submit.
# This is useful for setting default environmental settings.
# Example:
# spark. master                     spark://master:7077
# spark. eventLog. enabled          true
# spark. eventLog. dir              hdfs://namenode:8021/directory
# spark. serializer                 org. apache. spark. serializer. KryoSerializer
# spark. driver. memory             5g
# spark. executor. extraJavaOptions  - XX: + PrintGCDetails - Dkey = value - Dnumbers = " one two
three"
spark. default. parallelism 6
```

在文件中添加一行配置：spark. default. parallelism 6。重新启动脚本，加载文件，并查询加载后 RDD 的分区数。

注意：默认的并行度只是作为加载文件时分区数的最小值参考，实际的分块数由加载文件时的 Splits 数决定，即文件的 Blocks 数。也可以由加载时的 API 参数指定分区数。

二、Key - Value 元素类型的 RDD 的分区数解析

所有 RDD 都可以从 getPartitions 方法中找出分区数和父 RDD 的分区数间的依赖关系。同时，针对 Key - Value 元素类型的 RDD，还提供了精细控制的分区器 partitioner，可以通过设置分区器，来指定元素如何分区的策略，以及分区的个数。

当没有设置分区器时，默认使用 HashPartitioner，具体参见分区器中的默认分区器定义源码：

```
object Partitioner {
  /**
   * Choose a partitioner to use for a cogroup - like operation between a number of RDDs.
   *
   * If any of the RDDs already has a partitioner,choose that one.
   *
   * Otherwise,we use a default HashPartitioner. For the number of partitions,if
   * spark. default. parallelism is set,then we' ll use the value from SparkContext
   * defaultParallelism,otherwise we' ll use the max number of upstream partitions.
```

```
  *
  * Unless spark. default. parallelism is set,the number of partitions will be the
  * same as the number of partitions in the largest upstream RDD,as this should
  * be least likely to cause out - of - memory errors.
  *
  * We use two method parameters (rdd,others) to enforce callers passing at least 1 RDD.
  */
def defaultPartitioner(rdd:RDD[_],others:RDD[_] * ):Partitioner = {
  val bySize = (Seq(rdd) ++ others). sortBy(_. partitions. size). reverse
  for (r <- bySize if r. partitioner. isDefined) {
    return r. partitioner. get
  }
  if (rdd. context. conf. contains("spark. default. parallelism")) {
    new HashPartitioner(rdd. context. defaultParallelism)
  } else {
    new HashPartitioner(bySize. head. partitions. size)
  }
}
}
```

从代码中可以看到，当没有设置分区数时，会使用“spark. default. paralelism”属性配置的值作为默认的分区数值。

以 Key - Value 为特定类型的 PairRDDFunctions 类中 reduceByKey 方法构建 RDD 为例，参见 reduceByKey 代码：

```
/**
 * Merge the values for each key using an associative reduce function. This will also perform
 * the merging locally on each mapper before sending results to a reducer,similarly to a
 * "combiner" in MapReduce. Output will be hash - partitioned with the existing partitioner/
 * parallelism level.
 */
def reduceByKey(func:(V,V) => V):RDD[(K,V)] = {
reduceByKey(defaultPartitioner(self),func)
}
```

在调用基础的 reduceByKey 方法（另两个 API 会分别设置分区数或分区器）时，由于没有设置分区数或分区器，因此使用了默认的分区器，这里的默认分区器就是使用上面在 object Partitioner 中提到的 defaultPartitioner 方法。

三、源码解析的扩展

在源码解析过程中，可以进行一些扩展，比如之前的 MapPartitionsRDD 源码：

```
private[spark] classMapPartitionsRDD[U:ClassTag,T:ClassTag](
    prev:RDD[T],
    f:(TaskContext,Int,Iterator[T]) => Iterator[U],  // (TaskContext,partition index,iterator)
    preservesPartitioning:Boolean = false)
  extends RDD[U](prev) {

  override val partitioner = if (preservesPartitioning) firstParent[T]. partitioner else None
```

```
    override def getPartitions:Array[Partition] = firstParent[T]. partitions

    override def compute(split:Partition,context:TaskContext) =
      f(context,split. index,firstParent[T]. iterator(split,context))
}
```



一般 RDD 设置了分区器 partitioner 时，在 getPartitions 中都会通过该分区器，控制如何从依赖的各个父 RDD 中获取各个分区的数据，而在 MapPartitionsRDD 中，只是根据参数 preservesPartitioning 是否为 true 来控制是否设置 MapPartitionsRDD 的分区器 partitioner。

因此引出了以下问题：当 preservesPartitioning 控制当前 MapPartitionsRDD 的 partitioner 的设置，preservesPartitioning 的作用是什么？两种取值 true 或 false 具体应该在什么情况下？

问题的解析：当前 MapPartitionsRDD 的 getPartitions 方法中并没有依据 partitioner 记录分区的数据来源，但作为分区器，它不仅可以用于 RDD 分区获取其依赖的父 RDD 的数据上，还可以将该分区器传递到其子 RDD 中，在这里，主要的目的是后者，将分区器信息在 DAG 的 Lineage 上进行传递。

Section 2.3 RDD API 的应用案例与解析

这部分主要是解析 RDD 隐式转换后的类型的常用 API 应用案例。在解析过程中，首先描述 API 的功能及在官网 API 上的定义，并基于 API 的定义给出一些案例，对于比较重要的 API，会给出一些常用的应用场景说明。

RDD 是一个分布式的数据集，这里分布式的概念，我们可以从另一个角度来理解，就是对大数据集的操作可以转换为对分布式存放的小数据的操作。在 RDD 提供的各种 API 中，重点理解各个 API 的输入参数（包含函数参数）和返回值（注意各自对应的类型信息），对其中的函数参数，也是一样，要理解该函数的输入和返回值的具体含义（尤其是类型信息），理解这些内容后，结合分布式概念就可以完全理解该 API 的功能了。

理解 API 功能后，就是该是如何使用了，刚开始使用时，可以先构建出 API 的各个参数，尤其是函数参数，然后传入该 API 并执行，执行结果验证无误后，可以试着简化代码，多尝试这种从复杂到简化的过程后，这种简化的函数式的编程方式就很容易理解和使用了。

对某些复杂的 API，会先提供易于理解的代码编写风格，然后用简化的 Scala 函数式编程方式进行重新编写，使非函数式编程的开发者能更容易过渡到函数式风格的编写。

在解析 API 时，对 API 做了一些分类，包括针对 RDD/扩展 RDD 的 API、RDD 间/扩展 RDD 间的 API 等。RDD 间/扩展 RDD 间的 API 是指对两个或两个以上的 RDD 之间进行操作的 API。

这部分内容主要是给出在开发调试场景下，如何去实践案例，并在实践过程中加深对 RDDAPI 的理解的方法。

准备工作包括以下两部分：

一、使用交互式方式进行 API 实践及解析

启动交互式工具 spark-shell，解析 API 实践的代码及其输出结果。如何启动交互式工

具 spark – shell 请参考章节 2. 2. 2 交互式工具的启动。

二、控制日志信息输出

在解析过程中，为了专注于 API 的代码及其输出信息，在交互式工具 spark – shell 的控制界面上输入下面代码（仅针对“org. apache. Spark”部分的日志信息），提高日志输出等级或关闭日志信息。

```
import org. apache. log4j. {Level,Logger}
Logger. getLogger("org. apache. spark"). setLevel(Level. WARN)          // 设置级别为 WARN
    Logger. getLogger("org. apache. spark"). setLevel(Level. OFF)       // 关闭日志输出
```

2. 3. 1 如何查找 RDD API 的隐式转换

隐式转换函数为装载不同类型的 RDD 提供了相应的额外功能。通过查看 SparkContext 提供的隐式转换，就可以找到额外功能定义的类，并查看其对外提供的 API 接口了。

隐式转换后的类包括以下几种：

1）PairRDDFunctions：该扩展类中的方法中输入的数据单元是一个包含两个元素的元组结构。Spark 会把其中第一个元素当成 key，第二个当成 value。

2）DoubleRDDFunctions：这个扩展类包含了很多数值的聚合方法。如果 RDD 的数据单元能够隐式变换成 Scala 的 double 数据类型，则这些方法会非常有用

3）OrderedRDDFunctions：该扩展类的方法需要输入的数据是 2 元元组，并且 key 能够排序。

4）SequenceFileRDDFunctions：这个扩展类包含一些可以创建 Hadoop sequence 文件的方法。输入数据必须是 2 元元组。但需要额外考虑到元组元素能够转换成可写类型。

一、自定义隐式转换类的场景

自定义隐式转换类可以用于对现有的 RDD 功能进行扩展。

一般情况下，对功能进行扩展时，可以构建自己的工具类，对 RDD 进行操作，比如，我们定义一个过滤方法，然后通过该方法将 RDD 拆分成两个 RDD 时，就可以构建一个自己的 API，用类似于 List 类的 override def span(p:A => Boolean):(List[A],List[A])方法。我们可以用类似下面的方式实现：

```
package objectmytools {
  defspan(rdd:RDD[A],f:A => Boolean):(RDD[A],RDD[A]) = {
   (rdd. filter(f),rdd. filter(! f))
  }
  …
}
```

这里为了方便调用，将 Span 方法直接放 package object 里，使用时不需要再显式导入（即不需要再手动添加 import 语句）。

一个 span 调用就可以把传入的 RDD 根据过滤函数 f 进行拆分了。

如果想更加简便的使用自定义扩展功能的话，我们可以借鉴 Spark 对 RDD 额外功能的扩展方式，即隐式转换，来扩展针对我们自己特定的类型而提供的额外 RDD API。比如，我

们有一个类 MyClass，这时候就可以针对该类型提供一个隐式转换，以及扩展 API 的定义，具体实现方式可直接参考 Spark 扩展额外功能的源码，比如下面所示源码：

```
class MyClass(…) {…}
class MyClass RDDFunctions (self:RDD[MyClass]) extends Logging with Serializable {
…
}

object MyClass RDDFunctions{
implicit def rddToMyClassRDDFunctions(rdd:RDD[MyClass])
:MyClass RDDFunctions = {
new MyClass RDDFunctions(rdd)
}
}
```

下面简单介绍下扩展的 RDD 类的隐式转换部分的源码。

二、Spark 1.3 之前的版本

在对 RDD 进行各种处理的过程中，需要手动导入隐式转换语句，语句如下：

```
import org.apache.spark.SparkContext._
```

通过导入语句，可以将 Object SparkContext 中的隐式转换加载到当前作用域中。导入后才能支持除 RDD［T］提供的 API 之外的其他 API，通过在 Object SparkContext 中查找关键字“rddTo”可以搜到一系列的 RDD 转换定义，源码如下所示：

```
// The following deprecated functions have already been moved to' object RDD' to
// make the compiler find them automatically. They are still kept here for backward compatibility
// and just call the corresponding functions in' object RDD'.

@deprecated("Replaced by implicit functions in the RDD companion object. This is " +
  "kept here only for backward compatibility.","1.3.0")
def rddToPairRDDFunctions[K,V](rdd:RDD[(K,V)])
    (implicit kt:ClassTag[K],vt:ClassTag[V],ord:Ordering[K] = null) = {
  RDD.rddToPairRDDFunctions(rdd)
}
@deprecated("Replaced by implicit functions in the RDD companion object. This is " +
  "kept here only for backward compatibility.","1.3.0")
def rddToAsyncRDDActions[T:ClassTag](rdd:RDD[T]) = RDD.rddToAsyncRDDActions(rdd)
@deprecated("Replaced by implicit functions in the RDD companion object. This is " +
  "kept here only for backward compatibility.","1.3.0")
def rddToSequenceFileRDDFunctions[K <% Writable:ClassTag,V <% Writable:ClassTag](
rdd:RDD[(K,V)]) = {
  val kf = implicitly[K => Writable]
  val vf = implicitly[V => Writable]
  // Set the Writable class to null and' SequenceFileRDDFunctions' will use Reflection to get it
  implicit val keyWritableFactory = new WritableFactory[K](_ => null,kf)
  implicit val valueWritableFactory = new WritableFactory[V](_ => null,vf)
  RDD.rddToSequenceFileRDDFunctions(rdd)
}
```

```
@ deprecated("Replaced by implicit functions in the RDD companion object. This is " +
  "kept here only for backward compatibility. ","1.3.0")
def rddToOrderedRDDFunctions[K:Ordering:ClassTag,V:ClassTag](
rdd:RDD[(K,V)]) =
  RDD.rddToOrderedRDDFunctions(rdd)
@ deprecated("Replaced by implicit functions in the RDD companion object. This is " +
  "kept here only for backward compatibility. ","1.3.0")
def doubleRDDToDoubleRDDFunctions(rdd:RDD[Double]) = RDD.doubleRDDToDoubleRDDFunctions
(rdd)
@ deprecated("Replaced by implicit functions in the RDD companion object. This is " +
  "kept here only for backward compatibility. ","1.3.0")
def numericRDDToDoubleRDDFunctions[T](rdd:RDD[T])(implicit num:Numeric[T]) =
  RDD.numericRDDToDoubleRDDFunctions(rdd)
```

这段代码是基于 1.3 版本的，为了保证版本的后续兼容性，Spark1.3 版本中将这些方法标注为“@ deprecated”，即表示不建议继续使用这些方法。

三、Spark 1.3 版本

Spark 1.3 版本中，这部分隐式转换函数已经被移到 object RDD 中，使用时不需要再手动导入，而是由编译器自动去识别。

具体源码如下：

```
/**
 * Defines implicit functions that provide extra functionalities on RDDs of specific types.
 *
 * For example, [[RDD.rddToPairRDDFunctions]] converts an RDD into a
[[PairRDDFunctions]] for
 * key - value - pair RDDs, and enabling extra functionalities such as
[[PairRDDFunctions.reduceByKey]].
 */
object RDD {
  // The following implicit functions were in SparkContext before 1.3 and users had to
  //' import SparkContext._ to enable them. Now we move them here to make the compiler find
  // them automatically. However,we still keep the old functions in SparkContext for backward
  // compatibility and forward to the following functions directly.
  implicit def rddToPairRDDFunctions[K,V](rdd:RDD[(K,V)])
    (implicit kt:ClassTag[K],vt:ClassTag[V],ord:Ordering[K] = null):PairRDDFunctions[K,V]
 = {
    new PairRDDFunctions(rdd)
  }
  implicit def rddToAsyncRDDActions[T:ClassTag](rdd:RDD[T]):AsyncRDDActions[T] = {
    new AsyncRDDActions(rdd)
  }
  implicit def rddToSequenceFileRDDFunctions[K,V](rdd:RDD[(K,V)])
      (implicit kt:ClassTag[K],vt:ClassTag[V],
keyWritableFactory:WritableFactory[K],
                valueWritableFactory:WritableFactory[V])
    :SequenceFileRDDFunctions[K,V] = {
    implicit val keyConverter = keyWritableFactory.convert
```

```
      implicit val valueConverter = valueWritableFactory. convert
      new SequenceFileRDDFunctions( rdd,
keyWritableFactory. writableClass( kt) ,valueWritableFactory. writableClass( vt) )
    }
    implicit def rddToOrderedRDDFunctions[K:Ordering:ClassTag,V:ClassTag](rdd:RDD[(K,V)])
      :OrderedRDDFunctions[K,V,(K,V)] = {
      new OrderedRDDFunctions[K,V,(K,V)](rdd)
    }
    implicit def doubleRDDToDoubleRDDFunctions(rdd:RDD[Double]):DoubleRDDFunctions = {
      new DoubleRDDFunctions(rdd)
    }
    implicit def numericRDDToDoubleRDDFunctions[T](rdd:RDD[T])(implicit num:Numeric[T])
      :DoubleRDDFunctions = {
      new DoubleRDDFunctions(rdd. map(x => num. toDouble(x)))
    }
  }
```

对应这部分隐式转换后的类型，它的 API 可以直接在 RDD 实例中被应用到计算中。

2.3.2 RDD[T]的分区相关的 API

解析对 RDD 分区设置相关的 API，包括 repartition、coalesce。在解析过程中，通过解读源码，可以理解这两个 API 之间的关系。

RDD 的分区个数对应了该 RDD 处理时的并行度，在实践过程中，可以通过修改一个 RDD 的分区数来优化性能。

一、coalesce

1. 定义

```
def coalesce(numPartitions:Int,shuffle:Boolean = false)(implicit ord:Ordering[T] = null):RDD[T]
```

2. 功能描述

coalesce 对 RDD 数据重新分区，如果是减少分区，直接设置新的分区数即可，如果增加分区个数，需要设置 Shuffle 为 true，否则分区设置无效。

需要注意的是，即使是减少分区个数，如果设置了 Shuffle 为 true，在重新分区过程中也会产生 Shuffle 过程。

3. 示例

coalesce 带两个参数，根据以下两种情况的组合，来分析该方法的作用。

1）numPartitions：分区数的增加/减少。

2）shuffle：重分区过程中是否进行 shuffle 操作。

```
// 从 Scala 数据集构建 RDD,并设置分区数为 2
scala > valstrRdd = sc. parallelize ( List ("a","b","c","c","e") ,2)
strRdd:org. apache. spark. rdd. RDD[String] = ParallelCollectionRDD[11] at parallelize at <console>:
21
//查看 RDD 的分区数
scala > strRdd. partitions. size
res5:Int = 2
```

```
//查看 RDD 构建的 Lineage 关系图
scala > strRdd.toDebugString
res6:String = (2) ParallelCollectionRDD[11] at parallelize at <console>:21 []
//下面是重新设置分区的操作,通过设置不同的参数查看 API 的效果
scala > strRdd.coalesce(1).partitions.size
res7:Int = 1
scala > strRdd.coalesce(1).toDebugString
res8:String =
(1) CoalescedRDD[13] at coalesce at <console>:24 []
|  ParallelCollectionRDD[11] at parallelize at <console>:21 []
scala > strRdd.coalesce(1,true).partitions.size
res9:Int = 1
scala > strRdd.coalesce(1,true).toDebugString
res10:String =
(1) MapPartitionsRDD[21] at coalesce at <console>:24 []
|CoalescedRDD[20] at coalesce at <console>:24 []
|ShuffledRDD[19] at coalesce at <console>:24 []
 + -(2) MapPartitionsRDD[18] at coalesce at <console>:24 []
    |  ParallelCollectionRDD[11] at parallelize at <console>:21 []
scala > strRdd.coalesce(4).partitions.size
res11:Int = 2
scala > strRdd.coalesce(4,true).toDebugString
res12:String =
(4) MapPartitionsRDD[26] at coalesce at <console>:24 []
|CoalescedRDD[25] at coalesce at <console>:24 []
|ShuffledRDD[24] at coalesce at <console>:24 []
 + -(2) MapPartitionsRDD[23] at coalesce at <console>:24 []
    |  ParallelCollectionRDD[11] at parallelize at <console>:21 []
//没有指定 shuffle 为 true 时,设置更小的分区数不起作用
scala > strRdd.coalesce(4).partitions.size
res13:Int = 2
scala > strRdd.coalesce(4,true).toDebugString
res14:String =
(4) MapPartitionsRDD[31] at coalesce at <console>:24 []
|CoalescedRDD[30] at coalesce at <console>:24 []
|ShuffledRDD[29] at coalesce at <console>:24 []
 + -(2) MapPartitionsRDD[28] at coalesce at <console>:24 []
    |  ParallelCollectionRDD[11] at parallelize at <console>:21 []
```

4. 示例解析

首先获取 RDD 的分区数，当前分区数为 2，然后使用 coalesce 测试将分区数增加或减少后的分区个数，测试时，对 shuffle 参数也分别进行了设置。

需要注意的是，即使在减少分区数时，如果 shuffle 参数设置为 true，对应的 Lineage 关系图中就会有 ShuffledRDD，会存在 Shuffle 过程。

5. 应用场景

1）大数据集加载并过滤，过滤后每个分区的数据量非常小，这时候可以使用该 API，减少分区数，把小数据量的分区合并成一个分区。

2）当将小数据量文件保存到外部存储系统时，可以通过将分区数重设为 1，使得输出的数据在一个文件中，方便查看。

3）当分区数太低，导致 CPU 使用率过低时，可以借用该方法增加分区数，即增加处理的并行度，这时候可以提高 CPU 的使用率，继而提高性能。



二、repartition

1. 定义

```
def repartition(numPartitions:Int)(implicit ord:Ordering[T] = null):RDD[T]
```

2. 功能描述

也是对 RDD 数据重新分区，通过对源码的解析，可以看到它实际上是通过调用 coalesce 方法来实现的，但是在调用时，将 shuffle 的参数设置为 true，因此，在调用 repartition 方法时，会产生 shuffle 过程。

说明：通过以上解析可知，如果重新分区不需要 Shuffle 即可完成的话，应该直接调用 coalesce 方法来实现。

3. repartition 的源码

```
/**
 * Return a new RDD that has exactly numPartitions partitions.
 *
 * Can increase or decrease the level of parallelism in this RDD. Internally,this uses
 * a shuffle to redistribute data.
 *
 * If you are decreasing the number of partitions in this RDD,consider using`coalesce`,
 * which can avoid performing a shuffle.
 */
def repartition(numPartitions:Int)(implicit ord:Ordering[T] = null):RDD[T] = {
  coalesce(numPartitions,shuffle = true)
}
```

repartition 方法其实就是 coalesce（numPartitions，shuffle = true）方法的替换。

4. 示例

```
scala > valstrRdd = sc.parallelize( List("a","b","c","c","e"),2)
strRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[32] at parallelize at <console>:
22

scala > strRdd.partitions.size
res17:Int = 2

scala > strRdd.repartition(1).partitions.size
res18:Int = 1

scala > strRdd.repartition(1).toDebugString
res19:String =
(1)MapPartitionsRDD[40] at repartition at <console>:25 []
|CoalescedRDD[39] at repartition at <console>:25 []
```

```
|ShuffledRDD[38] at repartition at <console>:25 []
 +-(2)MapPartitionsRDD[37] at repartition at <console>:25 []
    |  ParallelCollectionRDD[32] at parallelize at <console>:22 []

//和 coalesce(4,true)效果是一样的
scala>strRdd.repartition(4).partitions.size
res20:Int=4

scala>strRdd.repartition(4).toDebugString
res21:String=
(4)MapPartitionsRDD[48] at repartition at <console>:25 []
|CoalescedRDD[47] at repartition at <console>:25 []
|ShuffledRDD[46] at repartition at <console>:25 []
 +-(2)MapPartitionsRDD[45] at repartition at <console>:25 []
    |  ParallelCollectionRDD[32] at parallelize at <console>:22 []
```

5. 示例解析

可以看到，使用 repartition 进行数据重新分区后得到的 RDD 都会有 ShuffledRDD 的父依赖。

6. 应用场景

由上面的分析可知，repartition 实际上是 coalesce（numPartitions，shuffle = true）的缩写，因此，coalesce 的应用场景也适用于该 API。

2.3.3 RDD[T]常用的聚合 API

详细解析 RDD 一些聚合、归并操作的 API，具体包括 aggregate、reduce、fold 等。

一、aggregate

1. 定义

```
def aggregate[U](zeroValue:U)(seqOp:(U,T)=>U,combOp:(U,U)=>U)(implicit arg0:ClassTag
[U]):U
```

2. 功能描述

aggregate 函数首先用初始值（zeroValue）和 seqOp 操作，将每个分区里面的元素进行聚合，对聚合后每个分区会返回一个类型为 U 的值，然后再用 combOp 函数将各个分区的返回值再次进行聚合。

3. 示例

本案例中 U 的类型为 List[String]，而 RDD 的类型 T 为 String。

```
scala>valzeroValue:List[String]=Nil
zeroValue:List[String]=List()
//在分区内部使用的归并函数,以 zeroValue 为初始值,作用在分区的各个元素上
//这里的效果是将分区元素归并到一个 List 中
scala>defseqOP(newValue:List[String],elemValue:String):List[String]={
     |elemValue::newValue
     |}
seqOP:(newValue:List[String],elemValue:String)List[String]
```



```
//对分区归并的结果再次进行归并的操作,分区归并的结果是 List[String]
//这里将各个分区归并结果再次归并成一个 List
scala > defcombOp(partA:List[String],partB:List[String]):List[String] = {
     |partA:::partB
     | }
combOp:(partA:List[String],partB:List[String])List[String]
scala > valstrRdd = sc.parallelize ( List ("a","b","c","c","e") ,2)
strRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[50] at parallelize at <console>:
22
scala > strRdd.aggregate(zeroValue)(seqOP,combOp)
res24:List[String] = List(e,c,c,b,a)
```

简化，去掉 zeroValue 的定义：

```
scala > strRdd.aggregate[ List[String]](Nil)(seqOP,combOp)
res25:List[String] = List(b,a,e,c,c)
```

进一步简化，去掉函数的定义：

```
scala > strRdd.aggregate[ List[String]](Nil)( (a,b) => b::a,(l,r) => l:::r )
res26:List[String] = List(b,a,e,c,c)
```

4. 示例解析

其中，当 Scala 编译器无法进行类型推导时，则需要指定类型，比如指定 U 为 [List[String]]。简化的代码比较难理解，可以通过 API 的签名，strRDD 元素的类型等自己推导，以加深理解。

通过 aggregate 操作，最终得到类型为 U 的值，和初始值 zeroValue 的类型相同，但不需要和 RDD 中元素类型一致。

因此，通过 aggregate 操作可以得到与 RDD 元素类型 T 不同的聚合类型 U。

二、reduce

1. 定义

```
def reduce(f:(T,T) => T):T
```

2. 功能描述

使用具有交换性和关联性的二进制操作，对 RDD 的元素进行归并。

3. 示例

```
scala > valreduceRdd = sc.parallelize ( List ("a","b","c") ,2)
reduceRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[51] at parallelize at <console
>:22
scala > reduceRdd.partitions.size
res28:Int = 2
scala > valintRdd = sc.parallelize ( List[Int] (1,2,3,4,5,6) ,2)
intRdd:org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[52] at parallelize at <console>:22
//对 RDD 的元素进行归并操作
//先在各个分区上进行归并,然后将归并结果再次进行相同的归并
//这里取元素的最大值,a 和 b 是 RDD 的两个元素(或者说数据单元,T 的实例)
scala > intRdd.reduce( (a,b) => if(a > b) a else b)
```

```
res29:Int =6
scala > import java. lang. Math
import java. lang. Math
scala > intRdd. reduce( (a,b) => Math. max(a,b))
res30:Int =6
scala > intRdd. reduce( Math. max _)
res31:Int =6
```

4. 示例解析

需要注意的是，归并 RDD 数据后得到的值的类型必须是和 RDD 元素的类型一致。

5. 应用场景

通常用于将 Driver Program 端的 Scala 数据集转换为 RDD，可以用于用户自己构建的数据集，也可以针对前面 RDD 的返回数据集，比如 collect 操作返回的数组数据集等，转换为 RDD 后，就可以使用 RDD 提供的丰富的 API 操作进行分布式计算了。

三、fold

1. 定义

```
def fold( zeroValue:T) ( op:(T,T) => T) :T
```

2. 功能描述

聚合每个分区的元素，然后使用具有关联性的操作，以及一个初始值，将每个分区聚合的结果进行归并。

给定的 op（t1，t2）操作运行修改第一个参数值，并返回其结果，这可以避免对结果值的内存分配，但不应该修改第二个参数值。

3. 示例

```
scala > valintRdd = sc. parallelize ( List[Int] (1,2,3,4,5,6) ,2)
intRdd:org. apache. spark. rdd. RDD[Int] = ParallelCollectionRDD[53] at parallelize at < console > :23
// RDD 本质上是一个数据集合,它的 fold 功能和 List 的 fold 功能是差不多的
//只是采用分布式方式进行
//对分区子集合(Iterator[T])进行 fold,然后对该结果的数据集再用操作参数进行归并
scala > intRdd. fold(0)((a,b) => if(a > b) a else b)
res32:Int =6
scala > import java. lang. Math
import java. lang. Math
scala > intRdd. fold(0)((a,b) => Math. max(a,b))
res33:Int =6
scala > intRdd. fold(0)(Math. max _)
res34:Int =6
```

4. 示例解析

fold 操作时，用于归并的初始值以及操作的返回值的类型，都必须和 RDD 元素的类型一致。

四、fold 与 reduce 操作的对比

fold 与 reduce 的一个重要差别在于，前者在“op:(T,T) => T”语句作用在一个分区的元素集合上时，提供了一个初始值。

2.3.4 DoubleRDDFunctions(self:RDD[Double])常用的 API



这个扩展类包含了很多数值的聚合方法。如果 RDD 的数据单元能够隐式变换成 Scala 的 Double 数据类型，这些方法在做数据解析、数据统计时会非常有用。

一、histogram

1. 定义

```
def histogram(buckets:Array[Double],evenBuckets:Boolean = false):Array[Long]
def histogram(bucketCount:Int):(Array[Double],Array[Long])
```

2. 功能描述

针对元素数据类型为 Double 的 RDD 统计直方图。支持两种方式进行统计：

1）一种是自定义分桶区间（左闭右开区间：[ )）：返回两个数组，一个是每个分桶边界值，另一个是各个分桶的统计数。

2）一种是设置分桶数进行平均分：返回各个分桶的统计数。

3. 示例

```
scala > valhgRDD = sc. parallelize(List(1. 1,1. 2,2. 1,2. 2,2,3,4. 1,4. 3,7. 1,8. 3,9. 3),2)
hgRDD:org. apache. spark. rdd. RDD[Double] = ParallelCollectionRDD[76] at parallelize at <console
>:37
scala > hgRDD. histogram(Array(0. 0,4. 1,9. 0))
res64:Array[Long] = Array(6,4)
//设置 evenBuckets 为 true 时,会采用常量时间内快速分桶方法,
//应该在分布比较均匀的情况下使用
scala > hgRDD. histogram(Array(0. 0,4. 1,9. 0),true)
res65:Array[Long] = Array(6,3)
scala > hgRDD. histogram(3)
res66:(Array[Double], Array[Long]) = (Array(1. 1, 3. 833333333333334, 6. 566666666666668,
9. 3),Array(6,2,3))
```

二、mean

1. 定义

```
def mean():Double
    defmeanApprox(timeout:Long,confidence:Double = 0. 95):PartialResult[BoundedDouble
```

2. 功能描述

求 RDD 元素的平均值，meanApprox 是计算近似的平均值。API 名字带 Approx 的都是近似计算。

3. 示例

```
scala > valdbRdd = sc. parallelize(List(1. 1,1. 2,2. 1,2. 2,2,3,4. 1,4. 3,7. 1,8. 3,9. 3),2)
dbRdd:org. apache. spark. rdd. RDD[Double] = ParallelCollectionRDD[83] at parallelize at <console
>:37
scala > dbRdd. mean
```

```
res69:Double = 4.063636363636364
scala > dbRdd.meanApprox(1,0.9)
res70: org.apache.spark.partial.PartialResult[org.apache.spark.partial.BoundedDouble] = (partial:[-Infinity,Infinity])
scala > dbRdd.meanApprox(1000,0.9)
res72: org.apache.spark.partial.PartialResult[org.apache.spark.partial.BoundedDouble] = (final:[4.064,4.064])
```

三、sampleStdev

1. 定义

```
def sampleStdev():Double
```

2. 功能描述

计算 RDD 元素的样本标准偏差（sample standard deviation）。

3. 示例

```
scala > valdbRdd = sc.parallelize(List(1.1,1.2,2.1,2.2,2,3,4.1,4.3,7.1,8.3,9.3),2)
dbRdd:org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[85] at parallelize at <console>:37
scala > dbRdd.sampleStdev
res73:Double = 2.9042288915554604
```

四、sampleVariance

1. 定义

```
def sampleVariance():Double
```

2. 功能描述

计算 RDD 元素的样本偏差。

3. 示例

```
scala > valdbRdd = sc.parallelize(List(1.1,1.2,2.1,2.2,2,3,4.1,4.3,7.1,8.3,9.3),2)
dbRdd:org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[87] at parallelize at <console>:37
scala > dbRdd.sampleVariance
res74:Double = 8.434545454545457
```

五、stats

1. 定义

```
def stats():StatCounter
```

2. 功能描述

RDD 元素的统计，包含平均值、标准偏差、最大值和最小值。

3. 示例

```
scala > val dbRdd = sc.parallelize(List(1.1,1.2,2.1,2.2,2,3,4.1,4.3,7.1,8.3,9.3),2)
dbRdd:org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[87] at parallelize at <console>:37
```

```
scala > dbRdd.stats
res75: org.apache.spark.util.StatCounter = (count: 11, mean: 4.063636, stdev: 2.769074, max: 9.300000, min: 1.100000)
```



六、stdev

1. 定义

```
def stdev():Double
```

2. 功能描述

计算 RDD 元素的标准偏差。

3. 示例

```
scala > val dbRdd = sc.parallelize(List(1.1,1.2,2.1,2.2,2,3,4.1,4.3,7.1,8.3,9.3),2)
dbRdd:org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[90] at parallelize at <console>:37
scala > dbRdd.stdev
res76:Double = 2.769073598704326
```

七、sum、sumApprox

1. 定义

```
def sum():Double
def sumApprox(timeout:Long,confidence:Double = 0.95):PartialResult[BoundedDouble]
```

2. 功能描述

RDD 元素的求和和近似求和。

3. 示例

```
scala > val dbRdd = sc.parallelize(List(1.1,1.2,2.1,2.2,2,3,4.1,4.3,7.1,8.3,9.3),2)
dbRdd:org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[94] at parallelize at <console>:37
scala > dbRdd.sum
res78:Double = 44.7
scala > dbRdd.sumApprox(1,0.9)
res79: org.apache.spark.partial.PartialResult[org.apache.spark.partial.BoundedDouble] = (partial: [-Infinity,Infinity])
scala > dbRdd.sumApprox(1000,0.9)
res80: org.apache.spark.partial.PartialResult[org.apache.spark.partial.BoundedDouble] = (final: [44.700,44.700])
```

八、variance

1. 定义

```
def variance():Double
```

2. 功能描述

求 RDD 元素的方差。

3. 示例

```
scala > valdbRdd = sc.parallelize(List(1.1,1.2,2.1,2.2,2,3,4.1,4.3,7.1,8.3,9.3),2)
dbRdd:org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[95] at parallelize at <console>:37
```

```
scala > dbRdd. variance
res82:Double = 7. 667768595041324
```

2. 3. 5 PairRDDFunctions[K,V]聚合相关的 API

详细解析 PairRDDFunctions 的一些聚合/归并操作的 API，包括 aggregateByKey、reduceByKey、foldByKey 等。

针对 Spark 1. 3 之前的版本，以下所有案例都需要先导入隐式转换，如下：

```
scala > import org. apache. spark. SparkContext. _
import org. apache. spark. SparkContext. _
```

但当前为 Spark 1. 3 版本，隐式转换的方法已经移入 RDD 伴生对象中，因此不再需要导入隐式转换。

一、aggregateByKey

1. 定义

```
def aggregateByKey[U](zeroValue:U)(seqOp:(U,V) => U,combOp:(U,U) => U)(implicit arg0:
ClassTag[U]):RDD[(K,U)]
def aggregateByKey[U](zeroValue:U,numPartitions:Int)(seqOp:(U,V) => U,combOp:(U,U) =>
U)(implicit arg0:ClassTag[U]):RDD[(K,U)]
def aggregateByKey[U](zeroValue:U,partitioner:Partitioner)(seqOp:(U,V) => U,combOp:(U,U) =
> U)(implicit arg0:ClassTag[U]):RDD[(K,U)]
```

2. 功能描述

aggregateByKey，顾名思义，和 RDD 的 aggregate 方法在逻辑上的功能是一致的，只是，这里聚合操作的对象由 RDD 分区的全部数据变成了 PairRDDFunctions 分区中按 key 分组后得到的 value 数据集。

即，RDD 的 aggregate 与 PairRDDFunctions 的 aggregateByKey 这两个 API 的差异点，就在于对分区中进行聚合时，聚合的目标数据集不同。

3. 示例

案例中实现作为 Key 值的单词的统计功能、合并功能，具体代码及结果如下：

```
scala > valzeroValue:List[String] = Nil
zeroValue:List[String] = List()
//这里的函数和 aggregate 都是一样的,只是作用在相同 Key 的元素上
scala > defseqOP(newValue:List[String],elemValue:String):List[String] = {
     |      elemValue::newValue
     | }
seqOP:(newValue:List[String],elemValue:String)List[String]
scala > defcombOp(partA:List[String],partB:List[String]):List[String] = {
     |      partA:::partB
     | }
combOp:(partA:List[String],partB:List[String])List[String]
//注意,之前的版本,隐式转换定义在 SparkContext 中,所以需要手动导入
//Spark 1. 3 版本中已经移到 RDD 中,不用导入也可以正确转换成 PairRDDFunctions
```

```
scala > import org. apache. Spark. SparkContext. _
import org. apache. Spark. SparkContext. _
scala > valkvRdd = sc. parallelize( List( (1," a" ), (1," b" ), (1," c" ), (2," b" ), (2," c" ), (3,"
d" ) ),2)
kvRdd:org. apache. Spark. rdd. RDD[ (Int,String) ] = ParallelCollectionRDD[0] at parallelize at < con-
sole > :24
scala > kvRdd. aggregateByKey[ List[ String] ] (Nil) ( (a,b) => b::a, (la,lb) => la:::lb). collect
res0:Array[ (Int,List[ String] ) ] = Array( (2,List( c,b) ), (1,List( c,b,a) ), (3,List( d) ) )
```



简化，去掉 zeroValue 的定义或 U 的类型指定：

```
scala > kvRdd. aggregateByKey[ List[ String] ] (Nil) ( seqOP,combOp). collect
res5:Array[ (Int,List[ String] ) ] = Array( (2,List( c,b) ), (1,List( c,b,a) ), (3,List( d) ) )
scala > kvRdd. aggregateByKey( zeroValue) ( seqOP,combOp). collect
res6:Array[ (Int,List[ String] ) ] = Array( (2,List( c,b) ), (1,List( c,b,a) ), (3,List( d) ) )
```

进一步简化，去掉函数的定义：

```
scala > kvRdd. aggregateByKey[ List[ String] ] (Nil) ( (a,b) => b::a, (l,r) => l:::r). collect
res7:Array[ (Int,List[ String] ) ] = Array( (2,List( c,b) ), (1,List( c,b,a) ), (3,List( d) ) )
```

4. 示例解析

查看 aggregateByKey 的签名，可以看出下面两个 aggregateByKey 比第一个增加了分区器信息的设置，包含修改分区器的分区个数，替换分区器。

为了重点解析 aggregateByKey 的功能，我们以第一个 aggregateByKey 方法进行案例解析，在案例解析之后，继续从源码角度进一步对这一类的 API 进行详细解析。

注意：但当前为 Spark 1.3 版本，不需要导入隐式转换。

5. 扩展内容

以 aggregateByKey 为例，介绍下对 API 功能的推导方法。通过对参数中各个函数进行解析，来理解该 API 的功能。这只是个人使用 API 时常用的一种推导方法，当遇到问题时，建议从源码角度去解析。

解析过程是通过 RDD 的类型、各个参数函数的签名，以及最终 API 的返回类型进行推导的，这种推导方式可以应用在各种 API 的解析上，这样可以避免阅读源码才能理解功能的尴尬，提高对 Spark API 的学习效率。

补充：当推导 A 方法时，如果已经熟悉 B 方法，同时 B 方法又调用了 A 方法时，可以结合 B 方法的理解进行推导。如果有兴趣，可以试试结合 aggregateByKey 方法的理解，来推导 combineByKey 方法。

以下是 aggregateByKey 方法的详细推导过程：

1）RDD 的元素类型为[K,V]，即 Key - Value 形式的二元组类型，其中 Key 的类型为 K，Value 的类型为 V。

2）由签名可知，aggregateByKey 方法的初始值 zeroValue 类型为 U，seqOp 定义为(U,V) => U，combOp 定义为(U,U) => U，最终返回 RDD 的元素类型为[K,U]。

3）因此，产生的类型变化是 V => U，即用初始值 zeroValue 来归并类型为 V 的值，得到类型 U 的值，对应操作类型为：(U,V) => X（X 表示未知类型），具有该签名的只有 seqOp:(U,V) => U 函数，因此第一步归并操作应该是 seqOp 对 zeroValue 和 RDD 元素的 Value

进行的。

4）由于 RDD 是分布式的数据集，所有的操作应该先针对分区进行，所以 seqOp 是在分区元素上递归地进行归并，即不断地进行(U,V) => U 操作，最终分区归并结果为类型 U 的值。

5）到这一步，可以知道各个分区都返回了 U 值，而 aggregateByKey 是对整个 RDD 进行的，也就是最终要合并各个分区的结果，这时候就是对 U 类型的元素集合进行合并，最终得到 U 类型的值。因此合并的操作定义应该是(U,U) => U，参数中只有 combOp 符合该定义。

通过以上解析，就可以知道，用 zeroValue 和 seqOp 对分区进行归并，然后用 combOp 再对分区归并的结果数据集再次进行归并。

二、combineByKey

1. 定义

```
def combineByKey[C](createCombiner:(V) => C,mergeValue:(C,V) => C,mergeCombiners:(C,C) => C):RDD[(K,C)]
def combineByKey[C](createCombiner:(V) => C,mergeValue:(C,V) => C,mergeCombiners:(C,C) => C,numPartitions:Int):RDD[(K,C)]
def combineByKey[C](createCombiner:(V) => C,mergeValue:(C,V) => C,mergeCombiners:(C,C) => C,partitioner:Partitioner,mapSideCombine:Boolean = true,serializer:Serializer = null):RDD[(K,C)]
```

2. 功能描述

在每个 partition 中先创建初始 combiner（createCombiner），然后将当前 RDD 的各个分区的数据单元逐个输入各个分区对应的 combiner 进行处理，在各个 partition 处理结束后，再在 mergeCombiner 中将各个 partition 的处理结果进行综合处理。

3. 示例

案例的类型信息：

1）K：Int。

2）V：String。

3）C：List［String］。

```
scala > valzeroValue:List[String] = Nil
zeroValue:List[String] = List()
//这是为 mergeValue 构建初始值的函数,对分区元素进行 mergeValue 操作时
//第一次 mergeValue 要使用该函数将分区的元素转换为目标类型
scala > defcreateCombiner(zeroValue:String) = List(zeroValue)
createCombiner:(zeroValue:String)List[String]
//下面的函数功能和 aggregateByKey 中的函数参数是类似的
//可以认为 combineByKey 是通用的
//可以提供一个从 RDD 元素转换为初始值的操作来代替一个 ZeroValue 初始值
//aggregateByKey 最终也是调用这个通用的 api 来实现的
scala > def mergeValue(newValue:List[String],elemValue:String):List[String] = {
     |        elemValue::newValue
     | }
mergeValue:(newValue:List[String],elemValue:String)List[String]
```



```
scala > def mergeCombiners(partA:List[String],partB:List[String]):List[String] = {
     |      partA:::partB
     | }
mergeCombiners:(partA:List[String],partB:List[String])List[String]
scala > val kvRdd = sc.parallelize(List((1,"a"),(1,"b"),(1,"c"),(2,"b"),(2,"c"),(3,"d")),2)
kvRdd:org.apache.spark.rdd.RDD[(Int,String)] = ParallelCollectionRDD[16] at parallelize at <console>:22
scala > var combRdd = kvRdd.combineByKey(createCombiner,mergeValue,mergeCombiners)
combRdd:org.apache.spark.rdd.RDD[(Int,List[String])] = ShuffledRDD[17] at combineByKey at <console>:32
scala > combRdd.collect
res13:Array[(Int,List[String])] = Array((1,List(c,b,a)),(2,List(c,b)),(3,List(d)))
```

进一步简化，去掉函数的定义：

```
scala > var combRdd = kvRdd.combineByKey(List(_),(a:List[String],b:String) => b::a,(l:List[String],r:List[String]) => l:::r)
combRdd:org.apache.spark.rdd.RDD[(Int,List[String])] = ShuffledRDD[19] at combineByKey at <console>:28
scala > combRdd.collect
res20:Array[(Int,List[String])] = Array((1,List(c,b,a)),(2,List(c,b)),(3,List(d)))
```

4. 示例解析

各个参数以及返回值的含义如下：

1）使用 createCombiner，处理各个分区的第一个元素，得到类型为 C 的新值。

2）使用 mergeValue，对各个分区的元素进行 merge，对由 createCombiner 得到的输出值，和分区元素值不断执行 mergeValue，最后得到分区的归并值，类型为 C。

3）使用 mergeCombiners，对上一步得到的各个分区的归并值，再次进行合并，得到类型为 C 的值。

4）最后，返回类型为[(K,C)]。

5. 应用场景

该 API 将大数据的处理转变为对小数据量的分区级别的处理，然后合并各个分区处理后再次进行聚合。在大数据量对处理的性能影响极大的情况下，这种先分区再合并的模式可以极大提高性能。尤其是在各个分区聚合后的数据量很小的情况下。

该应用场景适用于所有类似的聚合操作，比如调用了该方法的 aggregateByKey，只是各自聚合 API 使用了不同的参数，以及对输入输出类型的要求不同而已。

如果查看源码，可以看到内部有 mapPartitions 方法的调用。

三、foldByKey

1. 定义

```
def foldByKey(zeroValue:V)(func:(V,V) => V):RDD[(K,V)]
def foldByKey(zeroValue:V,numPartitions:Int)(func:(V,V) =>   V):RDD[(K,V)]
def foldByKey(zeroValue:V,partitioner:Partitioner)(func:(V,V) =>   V):RDD[(K,V)]
```

2. 功能描述

对每个分区元素，基于相同 key 值的 value 数据集合，进行 fold 操作。参数 zeroValue 为 fold 操作时使用的初始值。

3. 示例

```
//通过模式匹配,把不同情况下的两个字符串合并在一起
//这里可以增加 b 为""的 case,由于 RDD 元素都非空,这里仅考虑 zeroValue 为""
scala > def func(a:String,b:String):String = (a,b)match {
     |        case("",b) => b
     |        case(a,b) => s"$a $b"
     | }
func:(a:String,b:String)String
scala > val kvRdd = sc.parallelize(List((1,"a"),(1,"b"),(1,"c"),(2,"b"),(2,"c"),(3,"d")),2)
kvRdd:org.apache.spark.rdd.RDD[(Int,String)] = ParallelCollectionRDD[20] at parallelize at < console > :22
scala > kvRdd.foldByKey("")(func _).collect
res21:Array[(Int,String)] = Array((1,a b c),(2,b c),(3,d))
```

简化，去掉函数的定义：

```
scala > kvRdd.foldByKey(""){ case("",b) => b
     |                         case(a,b) => s"$a $b"
     |                       }.collect
res22:Array[(Int,String)] = Array((1,a b c),(2,b c),(3,d))
```

4. 示例解析

可以将 RDD 的 Value 类型 V 转换为其他类型 C。

四、reduceByKey

1. 定义

```
def reduceByKey(func:(V,V) => V):RDD[(K,V)]
def reduceByKey(partitioner:Partitioner,func:(V,V) => V):RDD[(K,V)]
def reduceByKeyLocally(func:(V,V) => V):Map[K,V]
```

2. 功能描述

对每个分区元素，基于相同 key 值的 value 数据集合，进行 func 操作。与 fold 相比，该方法没有提供 zeroValue 初始值。由于没有初始值，当某 key 的 value 只有一个值时，func 是不会执行的。

3. 示例

1）示例 1：单词统计。

```
scala > val kvRdd = sc.parallelize(List(("a",1),("b",1),("c",1),("b",1),("c",1),("d",1)),2)
kvRdd:org.apache.spark.rdd.RDD[(String,Int)] = ParallelCollectionRDD[54] at parallelize at < console > :25
//和 reduce 功能类似,只是作用在相同 key 值的元素上,这里的归并操作为求和操作
scala > kvRdd.reduceByKey(_ + _).collect
```

```
res35:Array[(String,Int)] = Array((a,1),(b,2),(c,2),(d,1))
```

2）示例 2：合并相同 Key 值的 value。

```
scala > def func(a:String,b:String):String = (a,b)match {
     |       case("",b) => b
     |       case(a,b) => s"$a $b"
     | }
func:(a:String,b:String)String
scala > val kvRdd = sc.parallelize(List((1,"a"),(1,"b"),(1,"c"),(2,"b"),(2,"c"),(3,"
d")),2)
kvRdd:org.apache.spark.rdd.RDD[(Int,String)] = ParallelCollectionRDD[56] at parallelize at < con-
sole > :25
scala > kvRdd.reduceByKey(func _).collect
res36:Array[(Int,String)] = Array((1,a b c),(2,b c),(3,d))
```



进一步简化，去掉函数的定义：

```
scala > kvRdd.reduceByKey{ case("",b) => b
     |                       case(a,b) => s"$a $b"
     |                      }.collect
res5:Array[(Int,String)] = Array((1,a b c),(2,b c),(3,d))
```

4. 示例解析

在 reduce 时，只能得到相同类型的结果。

2.3.6 RDD 相互间操作的 API

RDD 整合其他 RDD 的 API，cartesian、union 等。

一、cartesian

1. 定义

```
def cartesian[U](other:RDD[U])(implicit arg0:ClassTag[U]):RDD[(T,U)]
```

2. 功能描述

求两个 RDD 的笛卡尔积。

3. 示例

```
scala > val strRdd = sc.parallelize(List[String]("a","b","c","c","e"),2)
strRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[25] at parallelize at < console
 > :22
scala > val intRdd = sc.parallelize(List[Int](1,2,3,4,5,6),2)
intRdd:org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at parallelize at < console > :22
scala > val strRdd = sc.parallelize(List("a","b","c","c","e"),2)
strRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[27] at parallelize at < console
 > :22
scala > val intRdd = sc.parallelize(List(1,2,3,4,5,6),2)
intRdd:org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[28] at parallelize at < console > :22
//这里使用了操作符的调用方式,类似于 +、- 操作,不用 . 来调用函数
```

```
scala > val first = strRdd cartesian intRdd
first:org.apache.spark.rdd.RDD[(String,Int)] = CartesianRDD[29] at cartesian at <console>:26
scala > val second = intRdd cartesian strRdd
second:org.apache.spark.rdd.RDD[(Int,String)] = CartesianRDD[30] at cartesian at <console>:26
scala > first.collect
res24:Array[(String,Int)] = Array((a,1),(a,2),(a,3),(b,1),(b,2),(b,3),(a,4),(a,5),(a,
6),(b,4),(b,5),(b,6),(c,1),(c,2),(c,3),(c,1),(c,2),(c,3),(e,1),(e,2),(e,3),(c,
4),(c,5),(c,6),(c,4),(c,5),(c,6),(e,4),(e,5),(e,6))
scala > second.collect
res25:Array[(Int,String)] = Array((1,a),(1,b),(2,a),(2,b),(3,a),(3,b),(1,c),(1,c),(1,
e),(2,c),(2,c),(2,e),(3,c),(3,c),(3,e),(4,a),(4,b),(5,a),(5,b),(6,a),(6,b),(4,
c),(4,c),(4,e),(5,c),(5,c),(5,e),(6,c),(6,c),(6,e))
```

二、union

1. 定义

```
def union(other:RDD[T]):RDD[T]
```

2. 功能描述

两个 RDD 的联合操作，即合并两个 RDD 的元素到一个 RDD 中。

3. 示例

```
scala > valleftRdd = sc.parallelize(List("a","b","c"),2)
leftRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[31] at parallelize at <console
>:22
scala > val rightRdd = sc.parallelize(List("c","e"),1)
rightRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[32] at parallelize at <console
>:22
scala > leftRdd.union(rightRdd).collect
res26:Array[String] = Array(a,b,c,c,e)
scala > (leftRdd union rightRdd).collect
res27:Array[String] = Array(a,b,c,c,e)
```

4. 示例解析

在 union 操作中，是不会进行去重的。

查看 union 操作的分区变化，对两个 RDD 进行联合后，得到的新 RDD 的分区数为这两个 RDD 的分区数之和，代码如下：

```
//如果对分区感兴趣的话,可以通过下面这种方式
//查看各种转换操作后的分区数变化
//union 实际就是将父依赖 RDD 的所有分区合并
//成自己的各个分区,最终的分区和父依赖 RDD 的分区是一一对应的
scala > leftRdd.partitions.size
res4:Int = 2
scala > rightRdd.partitions.size
res5:Int = 1
//注意,leftRdd.union(rightRdd)和先定义一个 var a = leftRdd.union(rightRdd)
//然后使用的效果是一样的,如果没有缓存,都是从源数据重新计算 RDD 的
scala > leftRdd.union(rightRdd).partitions.size
res6:Int = 3
```



三、zip 家族的 API

定义

```
def zip[U](other:RDD[U])(implicit arg0:ClassTag[U]):RDD[(T,U)]
def zipPartitions[B,C,D,V](rdd2:RDD[B],rdd3:RDD[C],rdd4:RDD[D])(f:(Iterator[T],Iterator[B],Iterator[C],Iterator[D]) => Iterator[V])(implicit arg0:ClassTag[B],arg1:ClassTag[C],arg2:ClassTag[D],arg3:ClassTag[V]):RDD[V]
def zipPartitions[B,C,D,V](rdd2:RDD[B],rdd3:RDD[C],rdd4:RDD[D],preservesPartitioning:Boolean)(f:(Iterator[T],Iterator[B],Iterator[C],Iterator[D]) =>  Iterator[V])(implicit arg0:ClassTag[B],arg1:ClassTag[C],arg2:ClassTag[D],arg3:ClassTag[V]):RDD[V]
def zipPartitions[B,C,V](rdd2:RDD[B],rdd3:RDD[C])(f:(Iterator[T],Iterator[B],Iterator[C]) =>  Iterator[V])(implicit arg0:ClassTag[B],arg1:ClassTag[C],arg2:ClassTag[V]):RDD[V]
def zipPartitions[B,C,V](rdd2:RDD[B],rdd3:RDD[C],preservesPartitioning:Boolean)(f:(Iterator[T],Iterator[B],Iterator[C]) => Iterator[V])(implicit arg0:ClassTag[B],arg1:ClassTag[C],arg2:ClassTag[V]):RDD[V]
def zipPartitions[B,V](rdd2:RDD[B])(f:(Iterator[T],Iterator[B]) => Iterator[V])(implicit arg0:ClassTag[B],arg1:ClassTag[V]):RDD[V]
def zipPartitions[B,V](rdd2:RDD[B],preservesPartitioning:Boolean)(f:(Iterator[T],Iterator[B]) => Iterator[V])(implicit arg0:ClassTag[B],arg1:ClassTag[V]):RDD[V]
def zipWithIndex():RDD[(T,Long)]
def zipWithUniqueId():RDD[(T,Long)]
```

（一） zip

1. 功能描述

拉链操作，将两个 RDD 中第 i 个元素组成一个元组，形成 Key - Value 形式的二元组类型的 PairRDD。

2. 示例

```
scala> val kRdd  = sc.parallelize(1 to 4,2)
kRdd:org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[24] at parallelize at <console>:22
scala> val vRdd  = sc.parallelize("a b c d".split(" "),2)
vRdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[25] at parallelize at <console>:22
scala> kRdd.zip(vRdd).toDebugString
res27:String =
(2)ZippedPartitionsRDD2[26] at zip at <console>:27 []
 |  ParallelCollectionRDD[24] at parallelize at <console>:22 []
 |  ParallelCollectionRDD[25] at parallelize at <console>:22 []
scala> kRdd.zip(vRdd).collect
res28:Array[(Int,String)] = Array((1,a),(2,b),(3,c),(4,d))
```

zip 操作时，对应分区的个数需一致，否则会报以下错误：

```
//这里修改了分区数
scala> val kRdd  = sc.parallelize(1 to 2,1)
kRdd:org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[28] at parallelize at <console>:22
scala> kRdd.zip(vRdd).toDebugString
java.lang.IllegalArgumentException:Can't zip RDDs with unequal numbers of partitions
        at org.apache.spark.rdd.ZippedPartitionsBaseRDD.getPartitions(ZippedPartitionsRDD.scala:57)
        at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:219)
        at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:217)
```

```
            at scala.Option.getOrElse(Option.scala:120)
            at org.apache.spark.rdd.RDD.partitions(RDD.scala:217)
            at org.apache.spark.rdd.RDD.firstDebugString$1(RDD.scala:1479)
            at org.apache.spark.rdd.RDD.toDebugString(RDD.scala:1512)
……
```

zip 操作时，对应各个分区的元素个数需一致，否则会报以下错误：

```
scala> val kRdd   = sc.parallelize(1 to 8,2)
kRdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelize a
t <console>:22
scala> (kRdd zip vRdd).collect
15/04/30 00:43:05 ERROR Executor: Exception in task 0.0 in stage 2.0(TID 5)
org.apache.spark.SparkException: Can only zip RDDs with same number of elements
in each partition
            at org.apache.spark.rdd.RDD$$anonfun$zip$1$$anon$1.hasNext(RDD.scala:746
)
            at scala.collection.Iterator$class.foreach(Iterator.scala:727)
            at org.apache.spark.rdd.RDD$$anonfun$zip$1$$anon$1.foreach(RDD.scala:742
)
……
```

3. 示例解析

zip 方法可以将元素类型不同的 RDD 进行拉链操作，需要注意的是，当拉链操作时的两边的元素个数需要保持一致。

（二）zipPartitions

功能与 zip 类似，但是提供了更多方式的 zip。

以下面这个 API 进行案例解析：

```
def zipPartitions[B,C,V](rdd2: RDD[B],rdd3: RDD[C])(f: (Iterator[T],Iterator[B],Iterator[C])
 => Iterator[V])(implicit arg0: ClassTag[B],arg1: ClassTag[C],arg2: ClassTag[V]): RDD[V]
```

1. 功能描述

将当前 RDD 和 rdd2，rdd3 进行拉链操作，具体的操作由 f 指定。

2. 示例

```
scala> val a = sc.parallelize(0 to 4,2)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[31] at parallelize at <console>:22
scala> val b = sc.parallelize(List("a","b","c","d"),2)
b: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[32] at parallelize at <console>:22
scala> val c = sc.parallelize(List('!','@','#','$'),2)
c: org.apache.spark.rdd.RDD[Char] = ParallelCollectionRDD[33] at parallelize at <console>:22
//和 zip 功能是类似的,只是作用在两个要 zip 的分区数据集,即 Iterator 上
scala> def zipFunc(aIter: Iterator[Int], bIter: Iterator[String], cIter: Iterator[Char]): Iterator[String]
 = {
        |   var res = List[String]()
        |   while(aIter.hasNext&& bIter.hasNext&& cIter.hasNext){
        |     val x = aIter.next + "  " + bIter.next + "  " + cIter.next
        |     res::= x
```

```
        |      }
        |
        |      res. iterator
        | }
zipFunc:(aIter:Iterator[Int],bIter:Iterator[String],cIter:Iterator[Char])Iterator[String]
scala > a. zipPartitions(b,c)(zipFunc). collect
res32:Array[String] = Array(1 b @ ,0 a ! ,3 d $ ,2 c #)
scala > a. zipPartitions(b,c)(zipFunc). collect
res32:Array[String] = Array(1 b @ ,0 a ! ,3 d $ ,2 c #)
scala > a. zipPartitions(b,c)(zipFunc). toDebugString
res33:String =
(2)ZippedPartitionsRDD3[35] at zipPartitions at <console>:31 []
  |  ParallelCollectionRDD[31] at parallelize at <console>:22 []
  |  ParallelCollectionRDD[32] at parallelize at <console>:22 []
  |  ParallelCollectionRDD[33] at parallelize at <console>:22 []
```



3. 示例解析

前面提到过，RDD 是一个分布式的数据集，对它的操作是以分区为单位进行并行计算的。因此，这里提供的 f:(Iterator[T],Iterator[B],Iterator[C]) => Iterator[V])应该对应于拉链操作的三个 RDD 的各个分区的 Iterator[X]（X 表示 T、B、C）。

即对三个 RDD 的第 i 个分区进行 f 操作，得到新的分区的 Iterator[V]。

（三）zipWithUniqueId

1. 功能描述

当前 RDD 元素进行拉链操作时，操作对象为元素和元素的索引值，从 0 开始。

2. 示例

```
scala > val vRdd    = sc. parallelize("a b c d". split(" "),2)
vRdd:org. apache. spark. rdd. RDD[String] = ParallelCollectionRDD[36] at parallelize at <console>:22
scala > vRdd. zipWithUniqueId. collect
res34:Array[(String,Long)] = Array((a,0),(b,2),(c,1),(d,3))
scala > vRdd. zipWithUniqueId. toDebugString
res35:String =
(2)MapPartitionsRDD[38] at zipWithUniqueId at <console>:25 []
  |  ParallelCollectionRDD[36] at parallelize at <console>:22 []
```

分区个数不同时：

```
scala > val vRdd    = sc. parallelize("a b c d e". split(" "),2)
vRdd:org. apache. spark. rdd. RDD[String] = ParallelCollectionRDD[40] at parallelize at <console>:22
scala > vRdd. zipWithUniqueId. toDebugString
res36:String =
(2)MapPartitionsRDD[41] at zipWithUniqueId at <console>:25 []
  |  ParallelCollectionRDD[40] at parallelize at <console>:22 []
scala > vRdd. zipWithUniqueId. collect
res37:Array[(String,Long)] = Array((a,0),(b,2),(c,1),(d,3),(e,5))
```

3. 示例解析

该操作是对 RDD 元素及其标号进行拉链操作，因此对分区个数没有什么限制。

2.3.7 PairRDDFunctions[K,V]间的相关 API

一、Join 家族的 API

1. 定义

```
def fullOuterJoin[W](other:RDD[(K,W)],numPartitions:Int):RDD[(K,(Option[V],Option[W]))]
def fullOuterJoin[W](other:RDD[(K,W)]):RDD[(K,(Option[V],Option[W]))]
def fullOuterJoin[W](other:RDD[(K,W)],partitioner:Partitioner):RDD[(K,(Option[V],Option[W]))]
def leftOuterJoin[W](other:RDD[(K,W)],numPartitions:Int):RDD[(K,(V,Option[W]))]
def leftOuterJoin[W](other:RDD[(K,W)]):RDD[(K,(V,Option[W]))]
def leftOuterJoin[W](other:RDD[(K,W)],partitioner:Partitioner):RDD[(K,(V,Option[W]))]
def rightOuterJoin[W](other:RDD[(K,W)],numPartitions:Int):RDD[(K,(Option[V],W))]
def rightOuterJoin[W](other:RDD[(K,W)]):RDD[(K,(Option[V],W))]
def rightOuterJoin[W](other:RDD[(K,W)],partitioner:Partitioner):RDD[(K,(Option[V],W))]
def join[W](other:RDD[(K,W)],numPartitions:Int):RDD[(K,(V,W))]
def join[W](other:RDD[(K,W)]):RDD[(K,(V,W))]
def join[W](other:RDD[(K,W)],partitioner:Partitioner):RDD[(K,(V,W))]
```

2. 功能描述

对两个 RDD 进行 join 操作，包括全外联、左外联、右外联以及 join。

3. 示例

案例使用的类型信息：

1）leftRdd:RDD[String,Int] --(name,age)，即元素类型为元组，元组包含名字和年龄。

2）rightRdd:RDD[String,Char] --(name,gender:f/m)，即元素类型为元组，元组包含名字和性别，性别由 f 或 m 表示，分别对应 female 和 male。

```
scala> val leftRdd = sc.parallelize(List(("Tom",21),("Jerry",31),("Mary",23)))
leftRdd:org.apache.Spark.rdd.RDD[(String,Int)] = ParallelCollectionRDD[36] at parallelize at <console>:22
scala> val rightRdd = sc.parallelize(List(("Tom",'m'),("Mary",'f'),("Henry",'m')))
rightRdd:org.apache.Spark.rdd.RDD[(String,Char)] = ParallelCollectionRDD[37] at parallelize at <console>:22
scala> leftRdd.join(rightRdd).collect
res31:Array[(String,(Int,Char))] = Array((Mary,(23,f)),(Tom,(21,m)))
scala> leftRdd.fullOuterJoin(rightRdd).collect
res32:Array[(String,(Option[Int],Option[Char]))] = Array((Mary,(Some(23),Some(f))),(Jerry,(Some(31),None)),(Henry,(None,Some(m))),(Tom,(Some(21)
,Some(m))))
scala> leftRdd.leftOuterJoin(rightRdd).collect
res33:Array[(String,(Int,Option[Char]))] = Array((Mary,(23,Some(f))),(Jerry,(31,None)),(Tom,(21,Some(m))))
scala> leftRdd.rightOuterJoin(rightRdd).collect
res34:Array[(String,(Option[Int],Char))] = Array((Mary,(Some(23),f)),(Henry,(None,m)),(Tom,(Some(21),m)))
```

4. 示例解析

查看各操作后 RDD 元素类型的变化，如下所示：

1）fullOuterJoin:RDD[K,V] fullOuterJoinRDD[K,W] => RDD[K,(Option[V],Option[W])]。

2）leftOuterJoin:RDD[K,V] leftOuterJoinRDD[K,W] => RDD[K,(V,Option[W])]。

3）rightOuterJoin:RDD[K,V] rightOuterJoinRDD[K,W] => RDD[K,(Option[V],W)]。

4）Join:RDD[K,V] join RDD[K,W] => RDD[K,(V,W)]。

其中，当类型为 Option[V]时，表示该 V 类型的值有两种状态，Some[V]或 None，可以结合元素类型是否为 Option[V]来解析全外联、左外联、右外联操作的含义，即是不是包含了关联缺失的输出。

5. 扩展知识

Join 类型有：cross join、inner join、left outer join、right outer join、full outer join。

首先都是基于 cross join（笛卡尔乘积），然后是 inner join，在笛卡尔乘积的结果集中去掉不符合连接条件的行。left outer join 是在 inner join 的结果集上再加上左表中没被选上的行，同时行的右表部分的每个字段都用 NULL 填充。right outer join 是在 inner join 的结果集上再加上右表中没被选上的行，同时行的左表部分的每个字段都用全用 NULL 填充。

2.3.8 OrderedRDDFunctions[K,V,P <:Product2[K,V]]常用的 API

包含两个 API：sortByKey 和 repartitionAndSortWithinPartitions。

一、sortByKey

1. 定义

```
def sortByKey(ascending:Boolean = true,numPartitions:Int = self.partitions.size):RDD[(K,V)]
```

2. 功能描述

这个方法按 Key 进行排序，并输出新 RDD。

3. 示例

```
scala > val kvRdd = sc.parallelize(List((3,"a"),(7,"b"),(5,"c"),(3,"b"),(6,"c"),(9,"d")),3)
kvRdd:org.apache.Spark.rdd.RDD[(Int,String)] = ParallelCollectionRDD[50] at parallelize at <console>:22
scala > kvRdd.sortByKey().collect
res35:Array[(Int,String)] = Array((3,a),(3,b),(5,c),(6,c),(7,b),(9,d))
scala > kvRdd.sortByKey(true).collect
res36:Array[(Int,String)] = Array((3,a),(3,b),(5,c),(6,c),(7,b),(9,d))
scala > kvRdd.partitions.size
res37:Int = 3
scala > kvRdd.sortByKey(false,2).partitions.size
res38:Int = 2
```

二、repartitionAndSortWithinPartitions

1. 定义

```
def repartitionAndSortWithinPartitions(partitioner:Partitioner):RDD[(K,V)]
```

2. 功能描述

这个方法使用指定的 Partitioner 对 RDD 重新分区，并且在各个分区内按 key 对元素进行排序，然后输出新 RDD。

3. 示例

```
scala > val kvRdd = sc. parallelize(List((3,"a"),(7,"b"),(5,"c"),(3,"b"),(6,"c"),(9,"
d")),3)
kvRdd:org. apache. Spark. rdd. RDD[(Int,String)] = ParallelCollectionRDD[62] at parallelize at < con-
sole > :37
//导入分区器
scala > import org. apache. Spark. HashPartitioner
import org. apache. Spark. HashPartitioner
scala > val rpRdd = kvRdd. repartitionAndSortWithinPartitions(new HashPartitioner(2))
rpRdd:org. apache. Spark. rdd. RDD[(Int,String)] = ShuffledRDD[63] at repartitionAndSortWithinPar-
titions at < console > :39
//默认的 HashPartitioner(2)中是通过以 key 对分区数取模的
//所以奇数和偶数就分发到两个分区上了,在查看排序时需要注意如何分区
//可以引入第二个偶数 key,然后查看分区内的元素的排序
scala > rpRdd. collect
res55:Array[(Int,String)] = Array((6,c),(3,a),(3,b),(5,c),(7,b),(9,d))
scala > kvRdd. partitions. size
res56:Int = 3
scala > rpRdd. partitions. size
res57:Int = 2
//为了更方面查看分区内的排序,修改偶数 key 的元素
scala > valkvRdd = sc. parallelize(List((3,"a"),(7,"b"),(5,"c"),(4,"b
),(6,"c"),(9,"d")),3)
kvRdd:org. apache. Spark. rdd. RDD[(Int,String)] = ParallelCollectionRDD[33] at p
rallelize at < console > :23
scala >    val rpRdd = kvRdd. repartitionAndSortWithinPartitions(new HashPartitione
(2))
rpRdd:org. apache. Spark. rdd. RDD[(Int,String)] = ShuffledRDD[34] at repartition
ndSortWithinPartitions at < console > :25
//奇数和偶数两个分区都已经排序
scala > rpRdd. collect
res37:Array[(Int,String)] = Array((4,b),(6,c),(3,a),(5,c),(7,b),(9,d))
```

Section

2.4 Spark 应用程序构建

由于在 Windows 7 下比 Linux 系统的构建过程更容易出现问题，这里选择用 Windows 7 系统进行构建。

2.4.1 基于 SBT 构建 Spark 应用程序的实例



本节简单介绍了基于文本编辑工具编写代码，然后使用 SBT 进行 Spark 应用程序的构建的步骤。

当前环境：Windows 7、Cygwin，Cygwin 的安装可以参考网络资源，如果是在 Linux 环境下，直接打开终端进行相似的操作即可。

1. 首先，打开 Cygwin Terminal，进入应用程序所在的目录

```
lenovo@ lenovo - Pc  ~
 $cd/cygdrive/e/bd/project/
```

注意：目录要以/cygdrive 开头。

2. 以默认的格式生成目录结构

```
lenovo@ lenovo - PC/cygdrive/e/bd/project
 $mkdirTestProject
 $ls
SimpleAppProjectTestProject
lenovo@ lenovo - PC/cygdrive/e/bd/project
 $cdTestProject/
lenovo@ lenovo - PC/cygdrive/e/bd/project/TestProject
 $mkdir src
lenovo@ lenovo - PC/cygdrive/e/bd/project/TestProject
 $mkdir src/main
lenovo@ lenovo - PC/cygdrive/e/bd/project/TestProject
 $mkdir src/main/scala
lenovo@ lenovo - PC/cygdrive/e/bd/project/TestProject
 $find .
.
./src
./src/main
./src/main/scala
```

3. 进入 scala 目录，编辑文件 SimpleApp. scala

```
lenovo@ lenovo - PC/cygdrive/e/bd/project/TestProject
 $cd src/main/scala
lenovo@ lenovo - PC/cygdrive/e/bd/project/TestProject/src/main/scala
 $vimSimpleApp. scala
```

进入 vim 编辑界面后，输入 i，开始输入代码：

```
/* SimpleApp. scala  */
import org. apache. spark. SparkContext
import org. apache. spark. SparkContext. _
import org. apache. spark. SparkConf
objectSimpleApp {
  def main(args:Array[String]){
    val logFile = "YOUR_Spark_HOME/README. md" // Should be some file on your system
```

```
        val conf = newSparkConf( ). setAppName( "Simple Application" )
        val sc = newSparkContext( conf)
        val logData = sc. textFile( logFile,2). cache( )
        val numAs = logData. filter( line => line. contains( "a" ) ). count( )
        val numBs = logData. filter( line => line. contains( "b" ) ). count( )
    println( "Lines with a:%s,Lines with b:%s". format( numAs,numBs) )
      }
    }
```

编辑完成后，按【Esc】键，退出编辑状态，然后输入:wq 或:x 保存文件。

注：这部分代码来自官网案例。

4. 返回到应用程序目录，编辑 sbt 构建文件 simple. sbt

```
name: = "Simple Project"
version: = "1. 0"
scalaVersion: = "2. 10. 4"
libraryDependencies + = "org. apache. spark" %% "spark - core" % "1. 3. 0"
```

注意：构建文件 simple. sbt 必须放在 src 同级目录下，否则编译时会报如下错误：

```
[info] Compiling 1 Scala source to E: \bd\project\TestProject\target\scala - 2. 11\classes. . .
[error] E: \bd\project\TestProject\src\main\scala\SimpleApp. scala: 2: object apache is not a member
of package org
[error] import org. apache. spark. SparkContext
[error]              ^
[error] E: \bd\project\TestProject\src\main\scala\SimpleApp. scala:3:object apache is not a member
of package org
[error] import org. apache. spark. SparkContext. _
[error]              ^
[error] E: \bd\project\TestProject\src\main\scala\SimpleApp. scala:4:object apache is not a member
of package org
[error] import org. apache. spark. SparkConf
[error]              ^
[error] E: \bd\project\TestProject\src\main\scala\SimpleApp. scala:9:not found:type SparkConf
[error]     val conf = newSparkConf( ). setAppName( "Simple Application" )
[error]                   ^
[error] E: \bd\project\TestProject\src\main\scala\SimpleApp. scala:10:not found:type SparkContext
[error]     val sc = newSparkContext( conf)
[error]                 ^
[error] 5 errors found
[error] ( compile:compile) Compilation failed
[error] Total time: 9s,completed 2015 - 4 - 3 3: 06: 11
```

5. 查看目录结构

```
lenovo@ lenovo - PC/cygdrive/e/bd/project/TestProject
 $find .
.
. /simple. sbt
. /src
. /src/main
```

```
./src/main/scala
./src/main/scala/SimpleApp.scala
```

构建文件 simple.sbt 必须与 src 在同级目录下。目录结构完整，开始使用 SBT 进行构建。

6. 使用 sbt package 命令打包应用程序

```
lenovo@lenovo-PC /cygdrive/e/bd/project/TestProject
$sbt package
[info] Loading globalplugins from C:\Users\lenovo\.sbt\0.13\plugins
[info] Set current project toTestProject(in build file:/E:/bd/project/TestProject/)
[info] Updating {file:/E:/bd/project/TestProject/}testproject...
[info] Resolvingjline#jline;2.12 ...
[info] Done updating.
```

出现下面提示信息时，编译成功。

```
lenovo@lenovo-PC /cygdrive/e/bd/project/TestProject
$sbt package
[info] Loading globalplugins from C:\Users\lenovo\.sbt\0.13\plugins
[info] Set current project to Simple Project(in build file:/E:/bd/project/TestProject/)
[info] Updating {file:/E:/bd/project/TestProject/}testproject...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Compiling 1 Scala source to E:\bd\project\TestProject\target\scala-2.10\classes...
[info] Packaging E:\bd\project\TestProject\target\scala-2.10\simple-project_2.10-1.0.jar ...
[info] Done packaging.
[success] Total time:61 s,completed 2015-4-3 3:20:10
```

注意：要在应用程序所在的目录下执行该命令。

7. 查看生成的 jar 包

```
lenovo@lenovo-PC /cygdrive/e/bd/project/TestProject
$find target/scala-2.10/
target/scala-2.10/
target/scala-2.10/classes
target/scala-2.10/classes/SimpleApp$$anonfun$1.class
target/scala-2.10/classes/SimpleApp$$anonfun$2.class
target/scala-2.10/classes/SimpleApp$.class
target/scala-2.10/classes/SimpleApp.class
target/scala-2.10/simple-project_2.10-1.0.jar
```

2.4.2 基于 IDEA 构建 Spark 应用程序的实例

这部分内容详细描述了如何在 IDEA 中构建 Spark 应用程序，并通过关联 Spark 源码，为 Spark 应用程序的调试工作做准备。

一、准备工作

当前安装的 IntelliJ IDEA 版本为 14.0.1，低版本可能操作上会有些差异。IDEA 需要安装的插件，可以通过以下步骤来查看或安装。

1. 从 File 菜单进入 Plugins 设置界面

单击 File 菜单下的 Settings…命令，进入 IDEA 配置界面，如图 2. 30 所示。

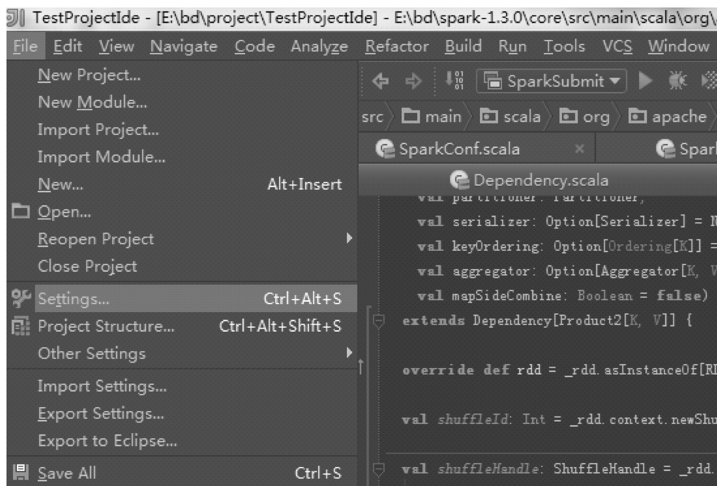


图 2. 30　IntelliJ IDEA 的 Settings 菜单

第一次打开 IDEA 或在 File 菜单下单击 Close Project 打开 IDEA 时，在出现的界面上可以打开配置页面。如图 2. 31 所示。

在这里单击下拉框后可以直接看到 Plugins，打开就可以进入插件管理界面，Configure 界面如图 2. 32 所示。



图 2. 31　IntelliJ IDEA 的 Configure 入口界面

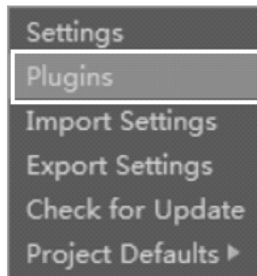


图 2. 32　IntelliJ IDEA 的 Plugins 配置选项

2. 安装插件

在 IDEA 的配置界面，即 Settings 界面上可以浏览已经安装的插件，以及查找插件仓库中的其他插件并安装，查找插件仓库中的插件过程如图 2. 33 所示。



图 2.33 IntelliJ IDEA 的 Plugins 配置界面

在 Plugins 界面中，可以看到已经安装的插件列表，这里查看是否已经安装了 SBT、SBT Executor 和 Scala 插件，如果还没有安装，单击 Browse repositories 按钮，可以浏览插件仓库，然后选择要安装的插件，并在界面右边单击 install 按钮即可。

二、构建 Spark 应用程序

构建 Spark 应用程序的详细步骤如下：

1. 创建 Project

在 File 菜单下单击 New Project…命令，创建一个新的 Project。创建一个 Project 的过程如图 2.34 所示。



图 2.34 IntelliJ IDEA 的创建新 Project 界面

2. 选择 Scala、SBT

Project 构建设置的过程如图 2.35 所示。

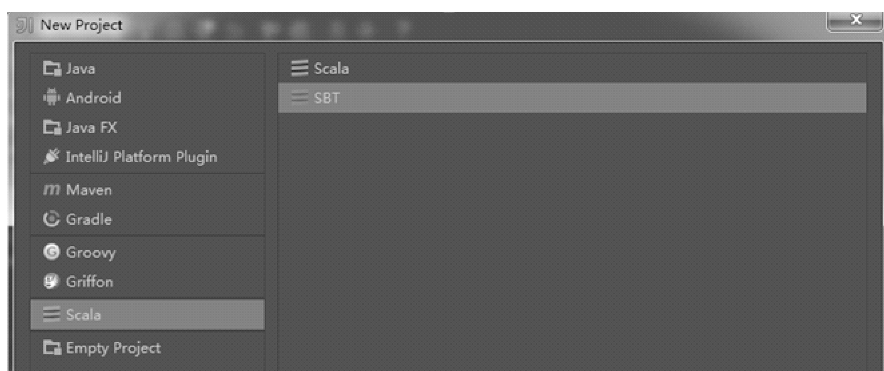


图 2. 35　IntelliJ IDEA 的创建 Project 界面

选择 Scala、SBT 后，单击 OK 按钮继续。

3. 设置 Project 名字，目录位置，以及选择 JDK 的版本信息

Project 的具体配置信息，包含名字、目录位置等的具体配置如图 2. 36 所示。

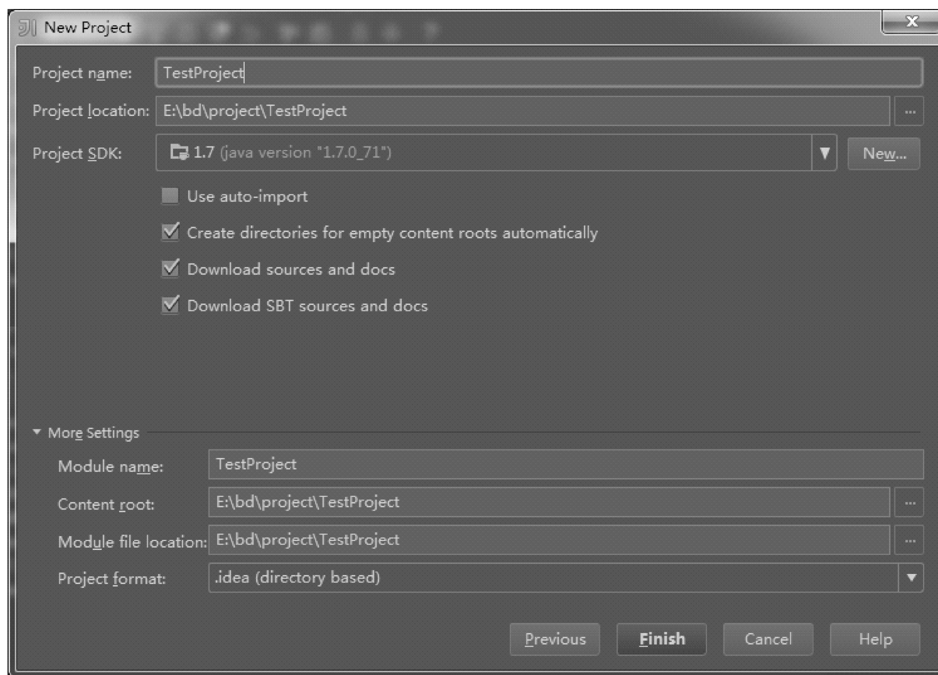


图 2. 36　IntelliJ IDEA 创建 Project 的配置界面

创建 Project 之前，先在本地安装 JDK（也可以后续在 File 菜单下的 Setting…中进行设置）。此处，单击 Finish 按钮继续。

4. 创建默认的目录结构

创建成功后，在 Project 中右键打开上下文菜单，在菜单中选择 Mark Directory As 命令，再选择 Generated Sources Root 命令，便自动构建了整个 Project 的目录结构，自动构建整个目录结构的操作流程如图 2. 37 所示。

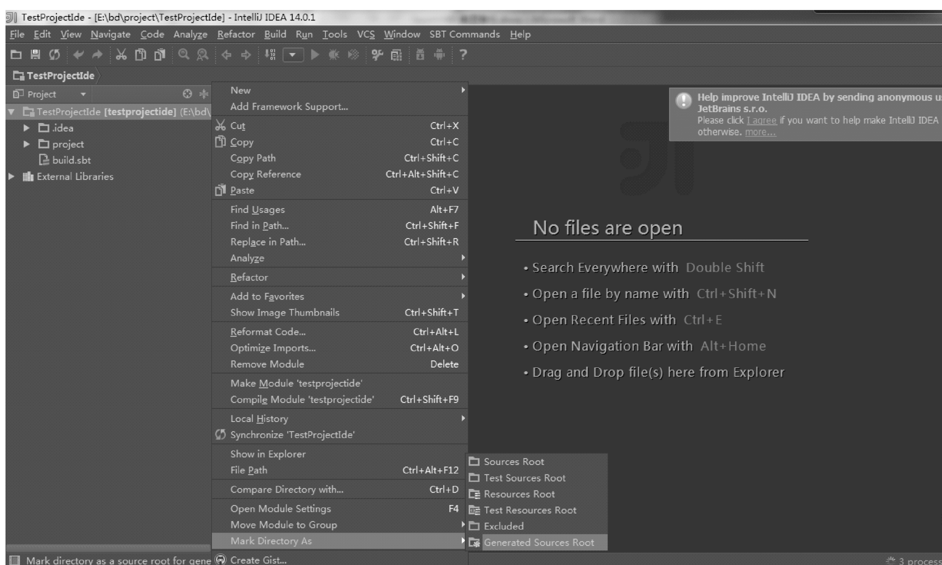


图 2.37　IntelliJ IDEA 的自动构建整个 Project 的目录结构菜单

创建过程中会进行编译，编译后的目录结构如图 2.38 所示。

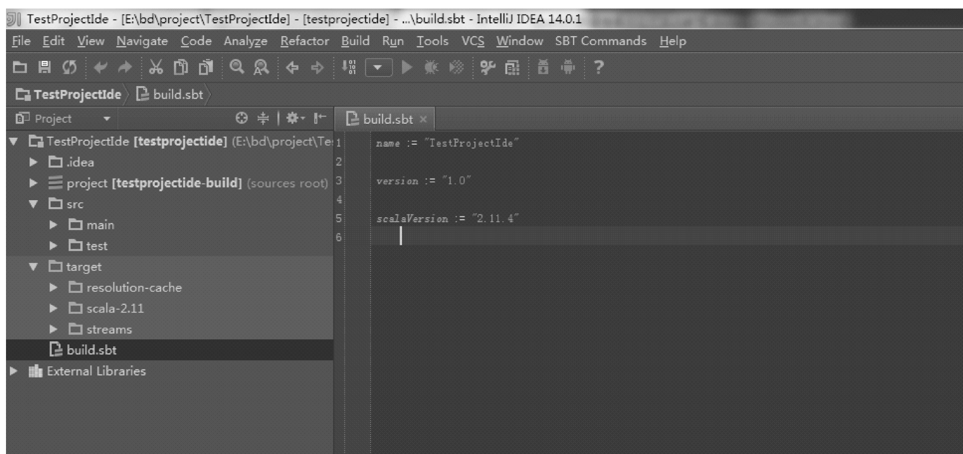


图 2.38　IntelliJ IDEA 的 SBT 构建文件

IDEA 已经构建了 src、src 下的 main 等各个子目录，同时包含了构建时生成的 target 目录，在这里可以看到，当前 IDEA 使用的 Scala 版本为 2.11.4。

5. 修改依赖的 Scala 版本

由于当前 Spark 1.3 是基于 Scala 2.10.4 版本构建的，考虑到 Scala 的二进制兼容性，构建的应用程序也需要保持 Scala 版本一致性，所以需要修改版本。

打开 build.sbt 文件，修改 ScalaVersion 为“2.10.4”，版本修改的文件内容如图 2.39 所示。

6. 执行 SBT Commands 下的 Compile 命令

在安装 SBT 相关的两个插件后，菜单中会出现 SBT Commands，新增的菜单及其具体子菜单如图 2.40 所示。

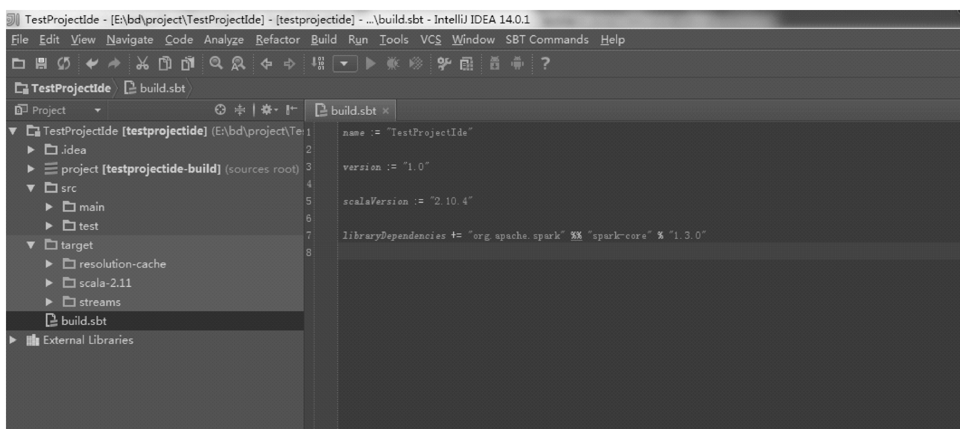


图 2.39　IntelliJ IDEA 的修改构建的 Scala 版本

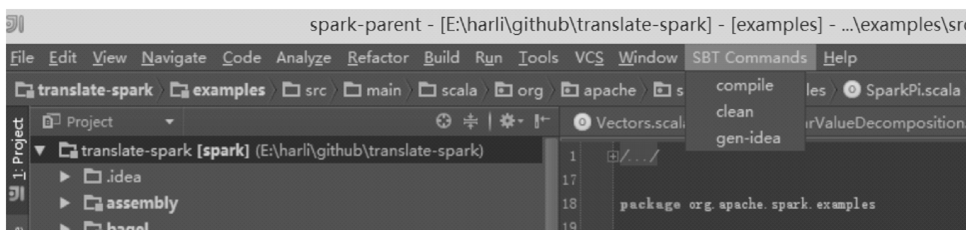


图 2.40　IntelliJ IDEA 插件的 SBT 命令

SBT Commands 包含以下几种命令：

1）compile：编译 Project。

2）clean：清理编译结果。

3）gen－idea：可以在 Windows 7 下使用，用于构建 IDEA 所需的工程信息。

运行这些命令后，可以在 SBT Execute Output 窗口查看具体过程，运行 compile 子菜单时的界面信息如图 2.41 所示。

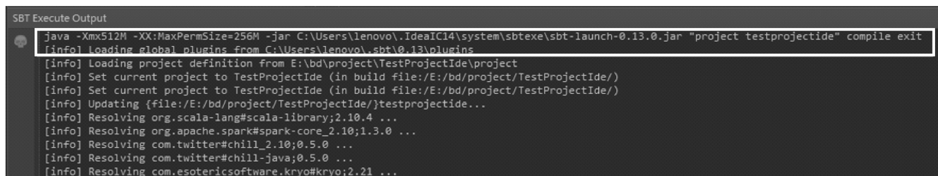


图 2.41　IntelliJ IDEA 中 sbt compile 执行过程

当出现图 2.42 中的界面信息时，执行成功。

由于之前修改了 Scala 的版本，对应构建后的 target 目录中会生成新的 Scala 版本对应的目录，结构如图 2.43 所示。

在 target 目录中编译的结果已经放到了 scala－2.10 目录。

7．在 src/main/scala 下创建 Scala Object

构建 Scala Object，在 scala 目录上单击右键，依次选择 New→Scala Class 命令，步骤如图 2.44 所示。

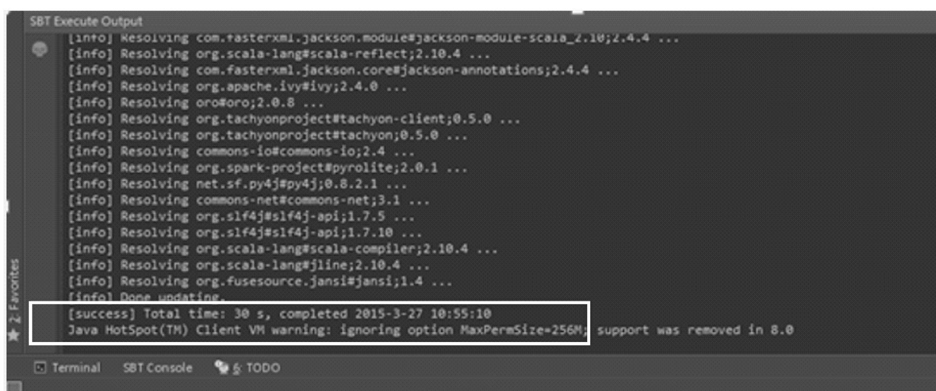


图 2.42　IntelliJ IDEA 中 sbt compile 执行结果

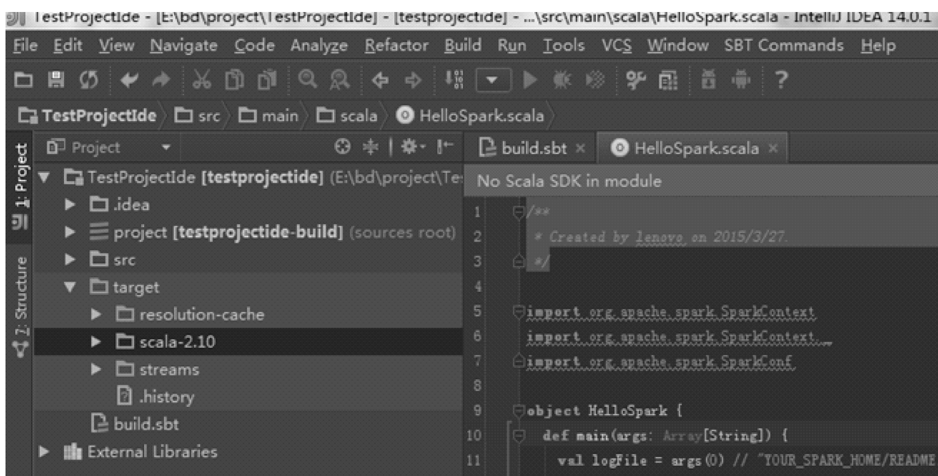


图 2.43　IntelliJ IDEA 中修改 Scala 版本并重新编译后的目录结构

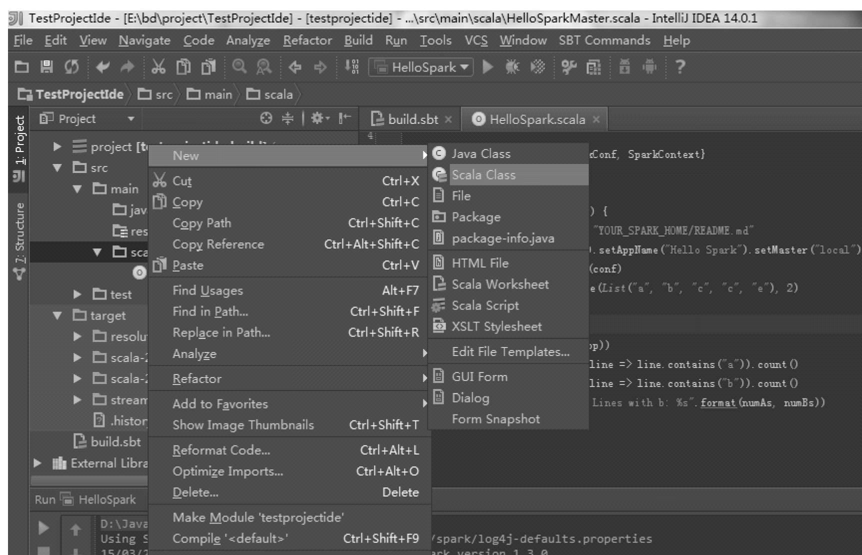


图 2.44　IntelliJ IDEA 中添加 Scala 类



在弹出窗口中，单击下拉页表框 Kind，选择 Object，如图 2.45 所示。

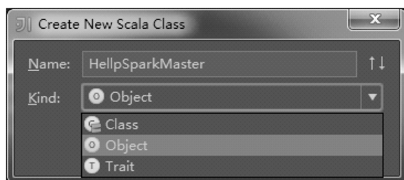


图 2.45　IntelliJ IDEA 中添加 Scala 类的类型选择

构建后，如图 2.46 所示，其中 HellpSparkMaster 位于 test 的 package 下：

在 Scala 中，package 的名字和目录不需要像 Java 那样，必须一致。但为了方便查看，建议使用像 Java 那样，保持两者的一致性，这里可以在 scala 目录上单击右键，然后构建一个 test 的 package，然后将 Scala Object 移到该 package 下。

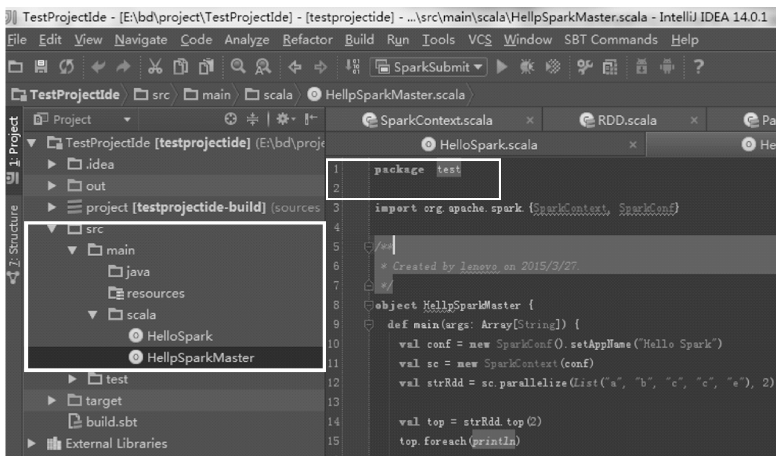


图 2.46　IntelliJ IDEA 中添加 Scala 类代码

8. 开始对应用程序进行打包：构建 jar 包

有两种方式可以使用，一种是在 IDEA 界面上使用 Terminal 窗口，在窗口中使用 sbt package 进行打包。如图 2.47 所示。

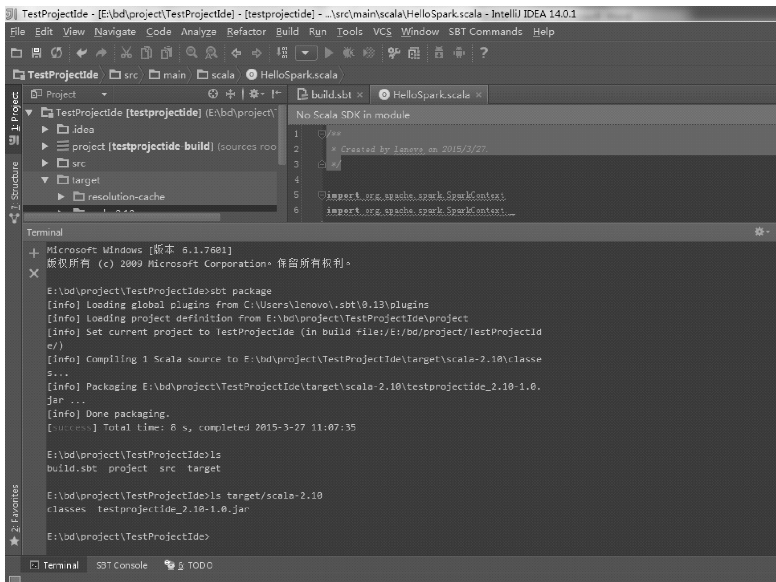


图 2.47　IntelliJ IDEA 的 sbt package 过程

Terminal 打开方式如图 2.48 所示，单击左下角箭头所指部分，选择 Terminal。

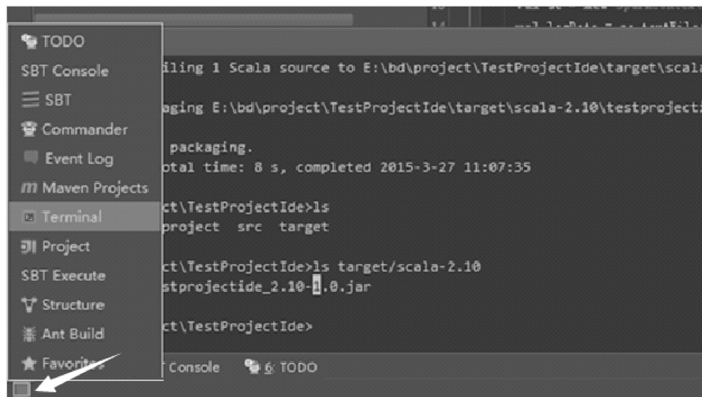


图 2.48　IntelliJ IDEA 的 Terminal 界面

这种方式构建的 jar 包和章节 2.4.1 基于 SBT 构建 Spark 应用程序的实例一样。

9. 添加依赖的 jar 包

虽然已经有了 sbt 构建文件，但 IDEA 不能识别 Spark 的类，在代码编辑窗口中还是有错误提示信息。可以手动将依赖的 jar 包添加进去，方法如下：

1）打开 Project Structure。即选择 File 菜单下的 Project Structure…命令，打开步骤如图 2.49 所示。

2）在 Libraries 中添加依赖的 jar 包。选中 Libraries 命令，单击“+”按钮，选择 Scala SDK 命令，具体步骤如图 2.50 所示。

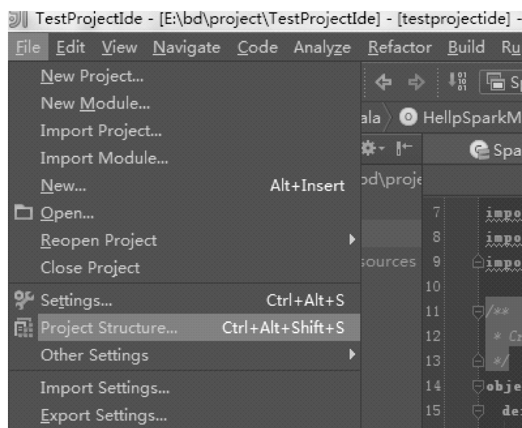


图 2.49　IntelliJ IDEA 的 Project Structure 菜单

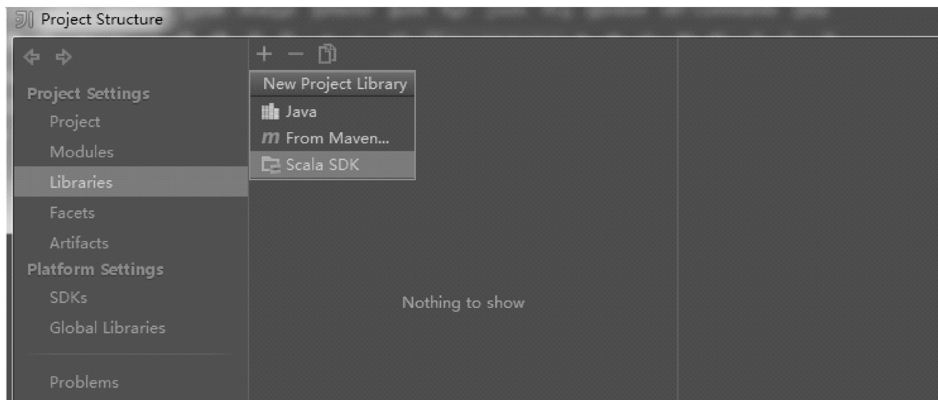


图 2.50　IntelliJ IDEA 中添加 Scala SDK 类库

3）在弹出窗口中选择 Scala 的 2.10.4 版本，版本选择界面如图 2.51 所示。

4）单击 OK 按钮，在弹出的窗口中选择当前的 Module，单击 OK 按钮，然后继续。

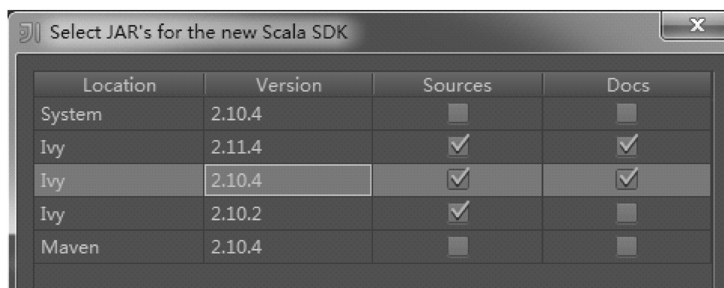


图 2.51　IntelliJ IDEA 中选择 Scala SDK 类库版本

5）继续单击“+”按钮，选择 Java 命令，添加 Spark 的编译后的 jar 包，添加依赖 jar 包界面如图 2.52 所示。

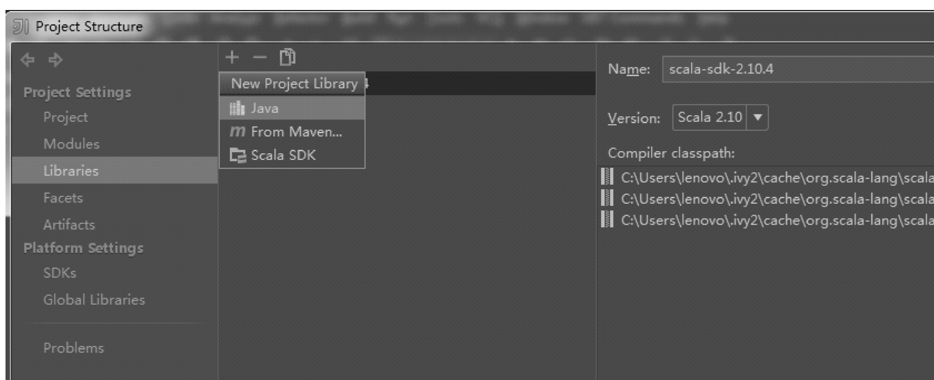


图 2.52　IntelliJ IDEA 中选择 Java 第三方类库

6）在 Spark 安装路径下添加所需的依赖 jar 包，如图 2.53 所示。

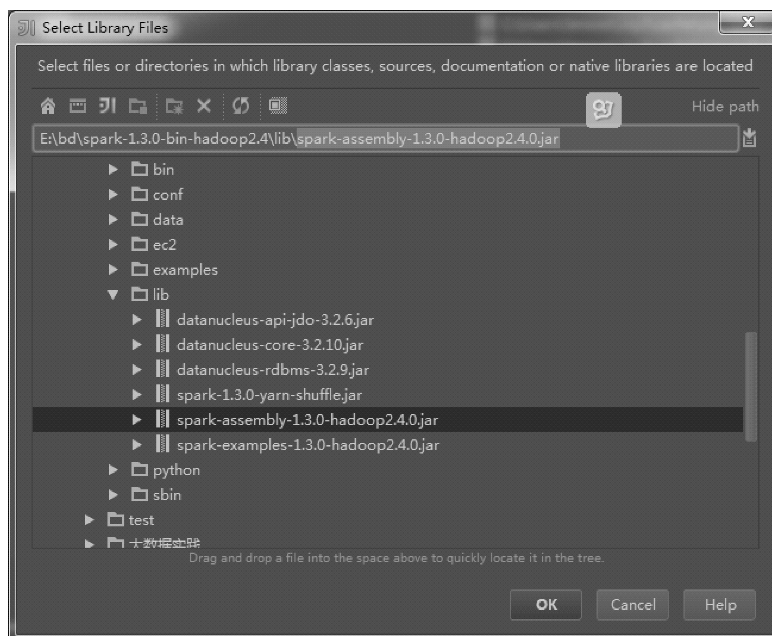


图 2.53　IntelliJ IDEA 中选择 Spark 依赖类库

7）这里选择最新的 Spark1.3 版本编译后的 Spark－assembly－1.3.0－hadoop2.4.0.jar 包。

8）单击 OK 按钮，在弹出的窗口中选择当前的 Module，单击 OK 按钮，完成依赖 jar 包的添加。

10. 通过 IDEA 的 Artifacts 构建 jar 包



1）打开 Project Structure。即选择 File 菜单下的 Project Structure…命令，在 Project Structure 窗口中选择 Artifacts 命令，Artifacts 配置如图 2.54 所示。

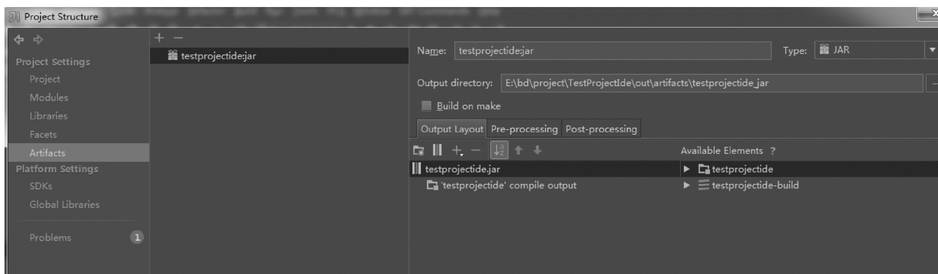


图 2.54　IntelliJ IDEA 中 Artifacts 配置界面

2）单击“＋”按钮，构建 jar 包，在窗口中填写相关信息，构建 jar 包的选择类型如图 2.55 中箭头所示。

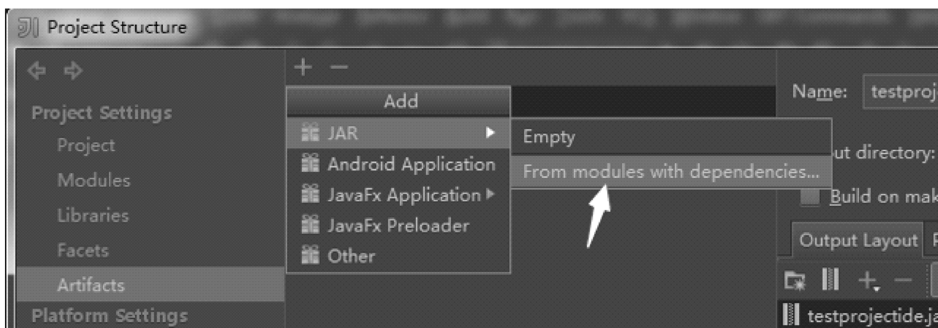


图 2.55　IntelliJ IDEA 中 Artifacts 配置

3）弹出窗口中选择 Module，当前 Module 的选择如图 2.56 中箭头所示。

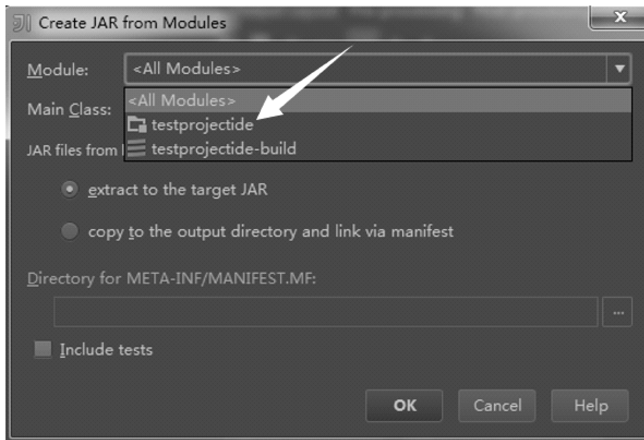


图 2.56　IntelliJ IDEA 中 Artifacts 的 Module 配置

4）将集群中已经部署的 jar 包去除。即选中 jar 包，单击“－”按钮，去除这部分依赖 jar 包，具体操作如图 2. 57 所示。

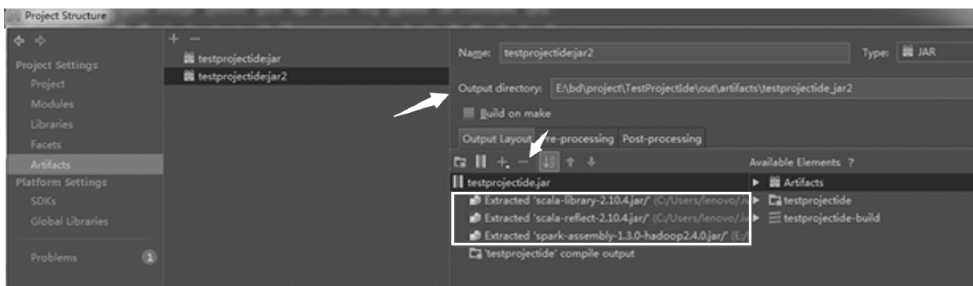


图 2. 57　IntelliJ IDEA 中 Artifacts 的配置

5）在 Artifacts 的配置界面中可以设置构建 jar 包的输出路径，可以在构建的 jar 包中去除依赖的 jar 包。单击 OK 按钮，继续在 IDEA 主菜单中选择 Build 下的 Build Artifacts…命令，操作步骤如图 2. 58 所示。

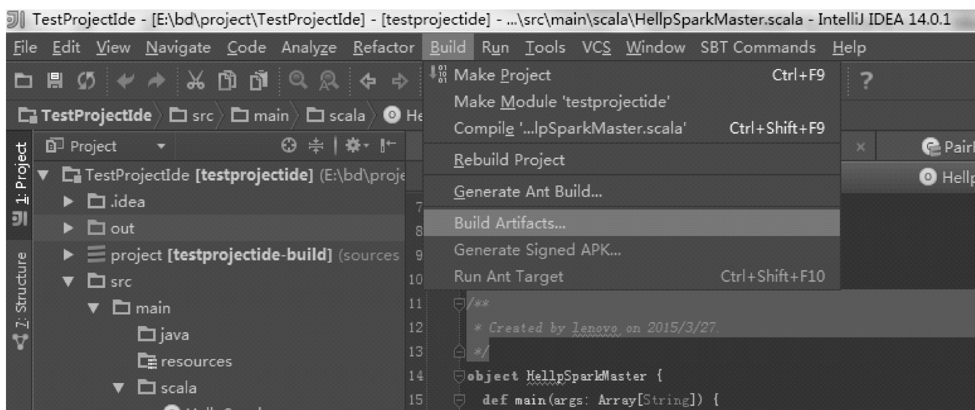


图 2. 58　IntelliJ IDEA 中 Artifacts 的构建菜单

6）构建后到输出路径上查看是否已经生成 jar 包，本例生成的结果文件如图 2. 59 所示。



图 2-59　IntelliJ IDEA 中 Artifacts 的 build 输出结果

到这一步，构建过程就完成了。

2.4.3 Spark 提交应用的调试实例

Local 方式调试请参考王家林老师的《大数据 Spark 企业级实践》。这里主要介绍 Master 集群方式下 Spark 应用程序的调试。

一、关联 Spark 源码

打开 Project Structure。选择 File 菜单下的 Project Structure…命令，Project Structure 界面如图 2.60 所示。

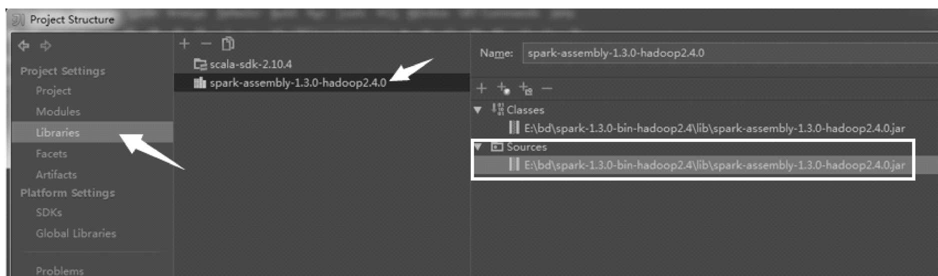


图 2.60　IntelliJ IDEA 中的 Project Structure…菜单

在图 2.60 中选中 Spark 的 jar 包 Spark – assembly – 1.3.0 – hadoop2.4.0，然后单击右面窗口中的“+”按钮，在弹出目录中选中 Spark 1.3 的源码目录，如图 2.61 所示。

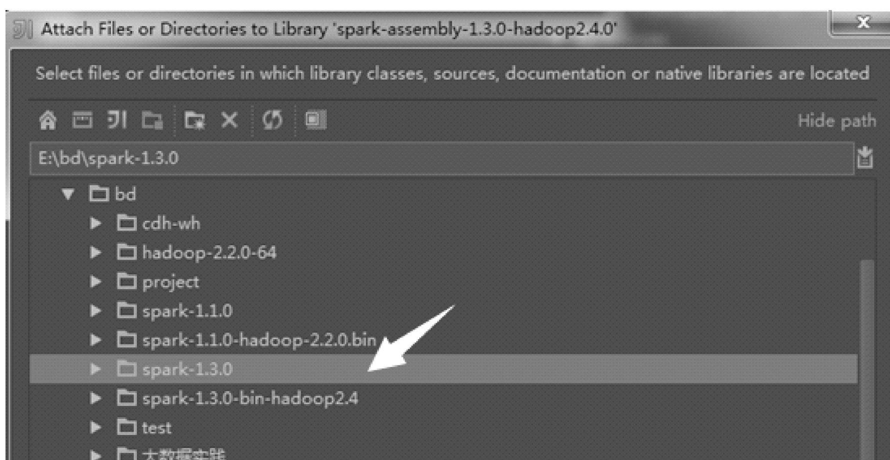


图 2.61　选择需要关联的源码目录

注意：有时候会弹出让你选择“+”的类型的窗口，这时候选择 Sources 就可以了。之后会弹出扫描进度的界面，如图 2.62 所示。

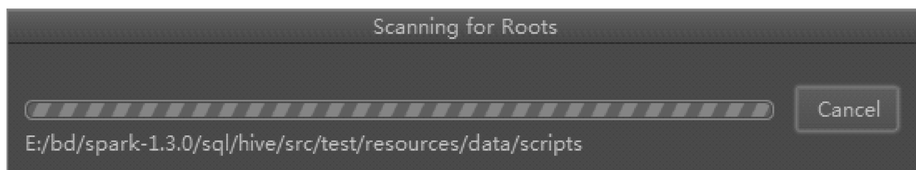


图 2.62　等待关联源码扫描过程

添加源码关联后，会自动开始扫描，等结束后会等弹出下面的窗口，如图 2.63 所示，默认全部选择即可，点击 OK 按钮。

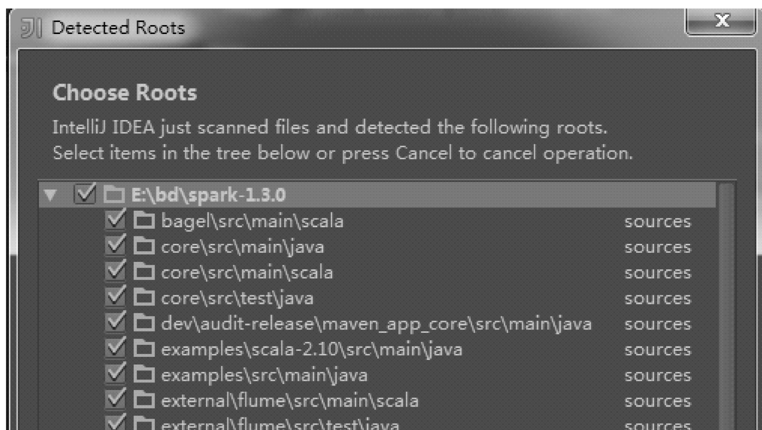


图 2.63　检测到的关联源码信息

如果编辑界面还是提示错误，单击 File 菜单下的 Invalidate Caches/Restart…命令，并重新启动 IDEA 即可，操作界面如图 2.64 所示。

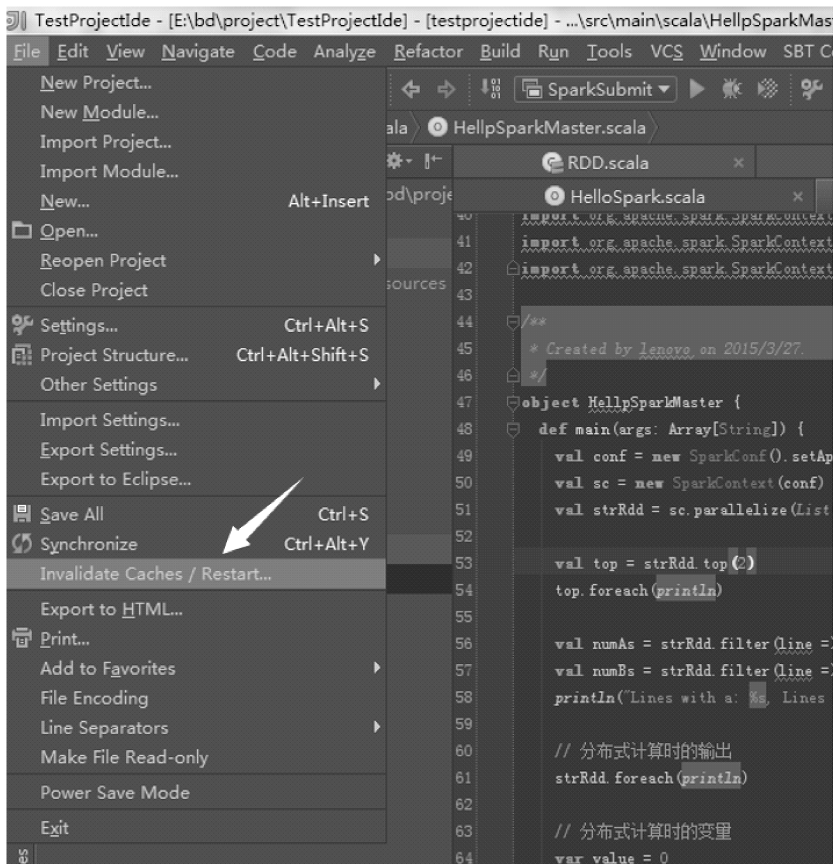


图 2.64　重新刷新缓存信息的菜单

到这一步，关联源码已经完成，可以开始调试 Spark 应用程序了。关联源码后，调试应用程序时，也可以同时调试 Spark 的源码。如果要远程调试 Spark 的各个组件，可以参考《Spark 学习笔记 . docx》的调试章节（https://github.com/harlixxy/StudyNotes/），笔记中同时提供了远程调试详细步骤的链接。

二、进行 Spark 应用程序的调试设置

单击 Run 菜单下的 Debug…命令，打开调试设置窗口，步骤如图 2.65 所示。

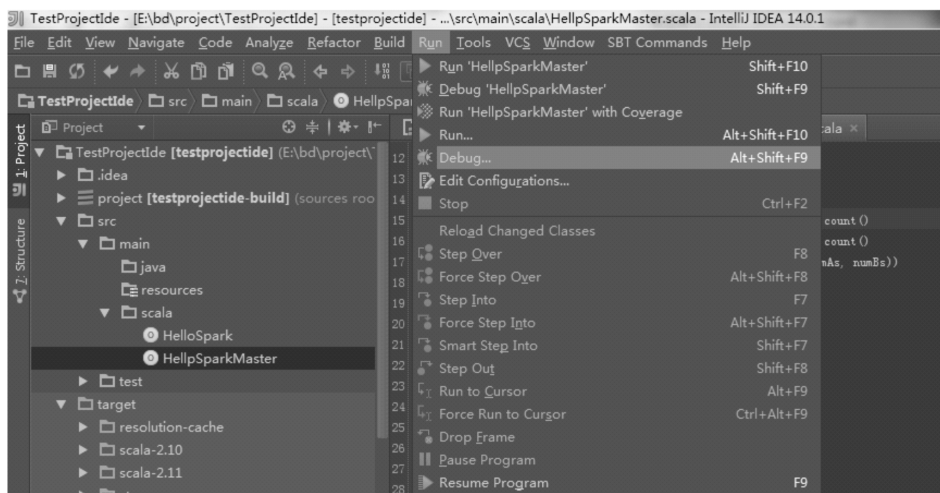


图 2.65　IntelliJ IDEA 中的 Debug…命令

在弹出窗口中，选择 Edit Configurations…命令，如图 2.66 所示。

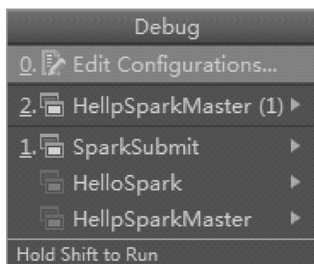


图 2.66　IntelliJ IDEA 中的调试编辑菜单

在左边窗口单击“+”按钮，添加一个应用程序（Application），添加后修改 Application 的配置信息，如图 2.67 所示。

为了模拟 spark - submit 提交部署应用程序，我们用 spark - submit 脚本最终执行的 SparkSubmit 作为调试入口。SparkSubmit 封装了集群应用的部署，通过 SparkSubmit 才能调试非 Local 的集群模式。进一步地，还可以通过设置环境变量来完全模拟 SparkSubmit 部署。

在图 2.67 中的右面窗口中：

1）Name：是添加的 Application 的名字。

2）Main class：是调试的入口类，设置为 org. apache. spark. deploy. SparkSubmit。

3）Program arguments：输入使用 spark - submit 脚本提交应用程序时所用的参数，包含



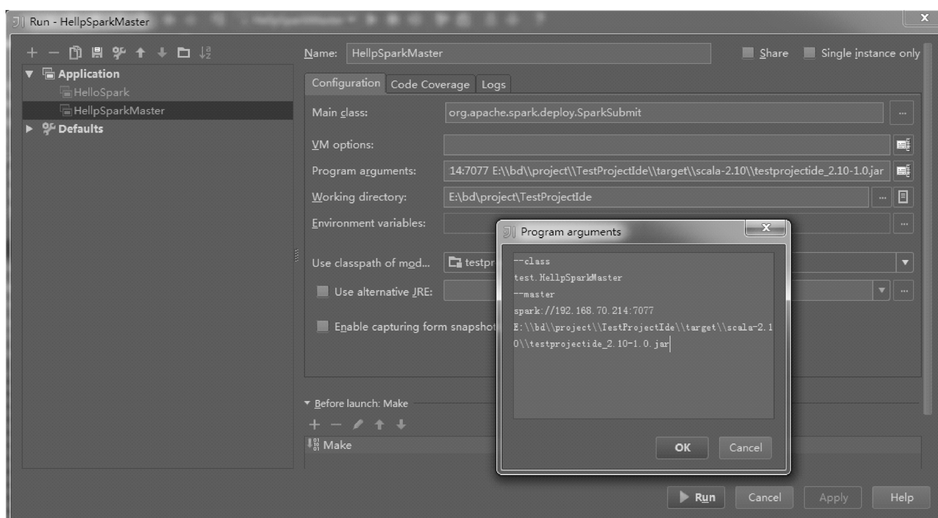


图 2.67　IntelliJ IDEA 中的调试时程序参数的设置界面

以下部分。

- --class：test.HellpSparkMaster，即要调试的应用程序。
- --master：Spark 集群的 MasterURL，必须和 Spark 集群 Web Interface（http://master:8080）上显示的一样。
- 最后是输入刚才构建的应用程序 jar 包的地址。

spark - submit 脚本提交应用程序最终也是将脚本的参数传入 SparkSubmit，因此这里使用的 Program arguments 和脚本的参数是一样的。

三、调试 Spark 应用程序

单击 Run 菜单下的 Debug…命令，打开调试设置窗口，在弹出窗口中选中刚才建立的调试 Application，这里对应刚才的调试编辑界面图 2.67 上的 Application 中的 HellpSparkMaster，如图 2.68 所示。

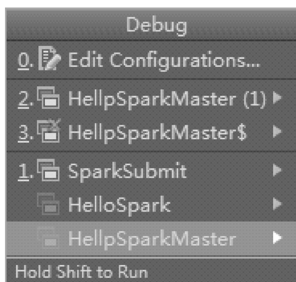


图 2.68　启动指定名字的应用程序调试

或在下拉框中选择调试的 HellpSparkMasterApplication，单击工具栏里的调试快捷按钮，如图 2.69 所示。

在应用程序 test.HellpSparkMaster 的 main 函数中设置断点，也可以按【Ctrl + N】组合键打开 SparkSubmit 类。在 SparkSubmit 中设置断点进行调试，调试窗口的内容如图 2.70 所示。

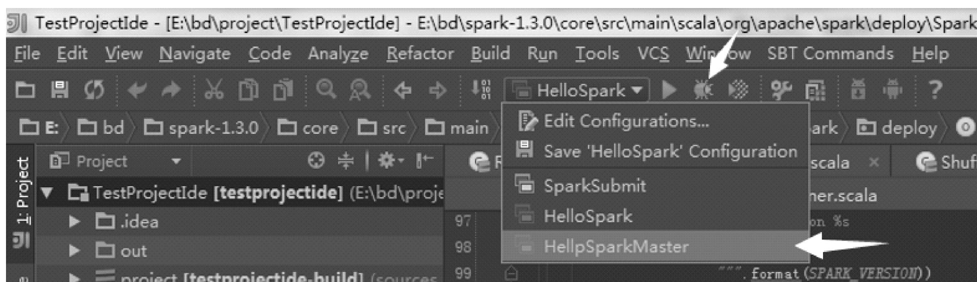


图 2.69　启动指定名字的应用程序调试的快捷方式

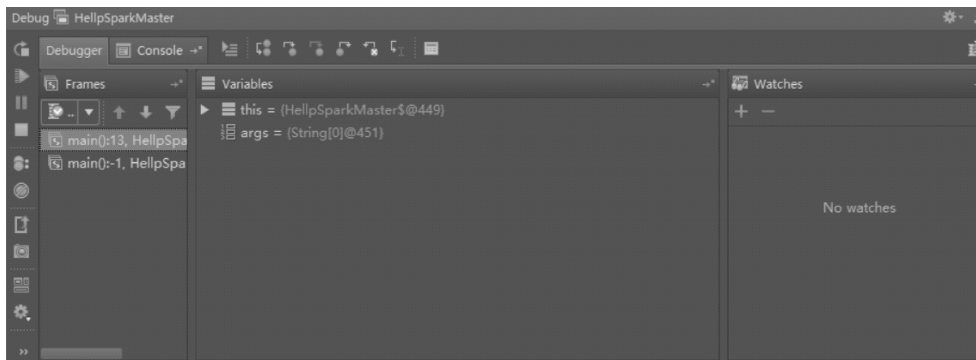


图 2.70　应用程序调试的界面信息

调试窗口中，Debugger 为调试窗口，可以看到调试信息，Console 为控制台窗口，可以看到控制台输出信息。控制台窗口的右边工具栏为调试工具栏，在 Variablies 窗口中可以查看当前程序的变量信息。

切换到 Console 输出窗口，查看 Console，即查看控制台输出的执行结果，下面是输出信息的截图。控制台 Console 输出信息如图 2.71 所示。

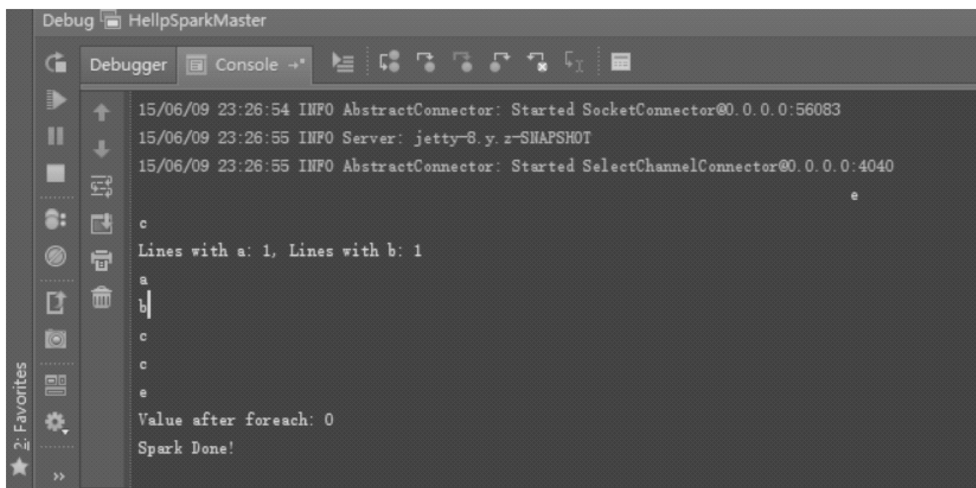


图 2.71　应用程序调试过程的部分输出信息

可以看到应用程序已经成功执行。下图是 Driver Program 的 Web Interface（http://dirver:4040）界面信息，应用程序执行结果如图 2.72 所示。

192.168.80.195:4040/executors/

Spark 1.3.0　Jobs　Stages　Storage　Environment　Executors

Executors (4)

Memory: 0.0 B Used (1235.6 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
0	wison223:36183	0	0.0 B / 265.0 MB	0.0 B	0	0	1	1	2.6 s	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
1	wison225:39538	0	0.0 B / 265.0 MB	0.0 B	0	0	5	5	10.0 s	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	wison215:55583	0	0.0 B / 265.0 MB	0.0 B	0	0	2	2	5.6 s	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
<driver>	192.168.80.195:54499	0	0.0 B / 440.6 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

图 2.72　应用程序调试时的 UI 界面

在窗口中可以看到刚才提交的应用程序。由于是在集群中提交，分布式计算，在 Task 执行时的输出信息是在 Executor 机器上输出的，可以在上面的界面 Logs 列进行查看。Logs 包括 stdout 和 stderr 两个文件（Executor 输出时重定向到这两个标准输出流），前者是标准输出信息，后者是标准错误。

下面是 stderr 文件内容的截图，内容如图 2.73 所示。

Spark 1.3.0　**stderr log page for app-20150327131847-0011/0**

Back to Master

Previous 0 B　Bytes 0 - 9003 of 9003

```
15/03/27 13:18:48 INFO executor.CoarseGrainedExecutorBackend: Registered signal handlers for [TERM, HUP, INT]
15/03/27 13:18:48 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/03/27 13:18:48 INFO spark.SecurityManager: Changing view acls to: harli,lenovo
15/03/27 13:18:48 INFO spark.SecurityManager: Changing modify acls to: harli,lenovo
15/03/27 13:18:48 INFO spark.SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(harli, lenovo); users w
ith modify permissions: Set(harli, lenovo)
15/03/27 13:18:49 INFO slf4j.Slf4jLogger: Slf4jLogger started
15/03/27 13:18:49 INFO Remoting: Starting remoting
15/03/27 13:18:49 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://driverPropsFetcher@wison223:22794]
15/03/27 13:18:49 INFO util.Utils: Successfully started service 'driverPropsFetcher' on port 22794.
15/03/27 13:19:16 INFO remote.RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.
15/03/27 13:19:16 INFO spark.SecurityManager: Changing view acls to: harli,lenovo
15/03/27 13:19:16 INFO spark.SecurityManager: Changing modify acls to: harli,lenovo
15/03/27 13:19:16 INFO spark.SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(harli, lenovo); users w
ith modify permissions: Set(harli, lenovo)
15/03/27 13:19:16 INFO remote.RemoteActorRefProvider$RemotingTerminator: Remote daemon shut down; proceeding with flushing remote transports.
15/03/27 13:19:16 INFO slf4j.Slf4jLogger: Slf4jLogger started
15/03/27 13:19:16 INFO Remoting: Starting remoting
15/03/27 13:19:16 INFO remote.RemoteActorRefProvider$RemotingTerminator: Remoting shut down.
15/03/27 13:19:16 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkExecutor@wison223:58944]
```

图 2.73　应用程序调试时 UI 界面的 stderr 文件内容

Section 2.5　移动互联网数据分析案例与解析

本章最后给出一个完整的基于移动互联网数据分析的案例与解析。通过对移动互联网数据的分析，了解移动终端在互联网上的行为以及各个应用在互联网上的发展情况等信息，具体包括对不同应用的使用统计、移动互联网上的日活跃用户（Day Active User，DAU）和月活跃用户数（Monthly Active Users，MAU）的统计，以及不同应用中的上行下行流量统计等分析。

2.5.1 移动互联网数据的准备

为了简化移动互联网数据的分析，这里提供简化模型后的数据信息，数据信息包含以下几个字段，字段名及其描述如下：

1）NodeID：基站 ID 信息。

2）CI：小区标识信息（CellIdentity）。

3）IMEI：国际移动电话设备识别码（International Mobile Equipment Identity）；手机串号。

4）APP：应用的名称。

5）Time：访问时间。

6）UplinkBytes：上行的字节数。

7）DownlinkBytes：下行的字节数。

移动互联网数据是移动终端对在互联网上的访问信息的记录，每一列分别对应前面的各个字段：NodeID，CI，IMEI，APP，Time，UplinkBytes 以及 DownlinkBytes。存放在../data/mobile.csv 文件中，各个字段模拟的数据如下所示（由于涉密原因，数据是人为构建的，真实数据会有一点差异，比如 Time 字段实际为系统时间戳，CI 同时会带有经纬度信息等）：

```
1,1,460028714280218,360,2015-05-01,7,1116
1,2,460028714280219,qq,2015-05-02,8,121
1,3,460028714280220,yy,2015-05-03,9,122
1,4,460028714280221,360,2015-05-04,10,119
2,1,460028714280222,yy,2015-05-05,5,1119
2,2,460028714280223,360,2015-05-01,12,121
2,3,460028714280224,qq,2015-05-02,13,122
3,1,460028714280225,qq,2015-05-03,1,1117
3,2,460028714280226,qq,2015-05-04,9,1118
3,3,460028714280227,qq,2015-05-05,10,120
1,1,460028714280218,360,2015-06-01,11,1118
1,2,460028714280219,qq,2015-06-02,2,1119
1,3,460028714280220,yy,2015-06-03,9,1120
1,4,460028714280221,360,2015-06-04,10,119
2,1,460028714280222,yy,2015-06-05,11,1118
2,2,460028714280223,360,2015-06-02,4,121
2,3,460028714280224,qq,2015-06-03,17,1119
3,1,460028714280225,qq,2015-06-04,18,119
3,2,460028714280226,qq,2015-06-05,19,1119
3,3,460028714280227,qq,2015-06-10,20,121
```

上传数据文件../data/mobile.csv 到 hdfs 的根目录下：

```
[harli@cluster04 cluster]$hdfs dfs -put data/mobile.csv /
[harli@cluster04 cluster]$hdfs dfs -ls /
Found 2 items
-rw-r--r--   1harli supergroup          828 2015-05-31 07:00 /mobile.csv
```

2.5.2 移动互联网数据分析与解析

移动互联网数据分析前需要先启动 spark - shell 交互命令：

```
bin/spark - shell -- executor - memory1g -- driver - memory 1g -- master spark://cluster04:7077
```

其中，cluster 04 为 spark 集群的 Master 节点。

启动后使用下面语句，去除过多的日志信息：

```
scala > import org. apache. log4j. {Level,Logger}
import org. apache. log4j. {Level,Logger}
scala > Logger. getLogger("org. apache. spark"). setLevel(Level. WARN)
scala > Logger. getLogger("org. apache. spark. sql"). setLevel(Level. WARN)
scala > Logger. getLogger("org. apache. hadoop. hive. ql"). setLevel(Level. WARN)
```

数据字段模型等变量的定义：

```
//定义当前移动互联网数据的字段列表
val fields = List("NodeID","CI","IMEI","APP","Time","UplinkBytes","DownlinkBytes")

//为了避免在每个 task 任务中传输 fields 信息,可以对其进行广播,代码如下
scala > valbcfields = sc. broadcast(fields)
bcfields:org. apache. spark. broadcast. Broadcast[List[String]] = Broadcast(4)
```

一、移动互联网数据的加载及预处理

这里分析的移动互联网数据已经是经过一定处理之后的数据，为了给出一些预处理（比如无效数据过滤）的案例，在各种统计分析之前先加载数据文件，并对文件进行简单处理。

首先加载文件，然后通过判断每行数据的字段个数，对访问记录的有效性进行判断。

```
//加载文件,并将每行记录以逗号分隔,最后根据字段个数进行过滤
scala > val mobile = sc. textFile("/mobile. csv"). map(_. split(",")). filter{case line if(line. length !
= bcfields. value. length) => false
     | case _ => true
     | }
mobile:org. apache. spark. rdd. RDD[Array[String]] = MapPartitionsRDD[57] at filter at <console
>:26
```

如果有多个有效性过滤条件，尽可能在一个 API 中，比如这里的 filter 中的判断函数。因为每个对分区进行操作的 API，都对应了一次 Iterator 的迭代，当分区记录比较多时，多次迭代会对性能会有一定影响。

二、不同应用使用情况的统计

对移动互联网数据的不同应用使用情况的统计，可以简化为对 APP 字段访问次数的简单统计，统计代码如下：

```
scala > mobile. map(x => (x(bcfields. value. indexOf("APP")),1)). reduceByKey(_ + _). map(x =>
(x. _2,x. _1)). sortByKey(false). map(x => (x. _2,x. _1)). collect
res45:Array[(String,Int)] = Array((qq,10),(360,6),(yy,4))
```

首先，第一个 map 将移动互联网数据转换为 org. apache. spark. rdd. RDD[(String,Int)]类型，其中，String 对应 APP 的信息，Int 对应使用的计数信息；然后使用 reduceByKey 对 APP 进行计数统计；最后通过 map 交换 key 与 value，并使用 sortByKey 进行排序，这里 sortByKey 的参数为 false，表示计数从大到小进行排序。

为了方便查看，这里通过 collect 在界面上显示结果，在实际的应用中，可以保存到文件中，如下所示：

```
scala > mobile. map(x => (x(bcfields. value. indexOf("APP")),1)). reduceByKey(_ + _). map(x =>
(x. _2,x. _1)). sortByKey(false). map(x => (x. _2,x. _1)). repartition(1). saveAsTextFile("/result/
appstat1")
```

在保存文件时，通过 repartition(1)指定分区个数为 1，可以将所有内容保存到一个文件中。文件保存的结果如下：

```
[harli@ cluster04 cluster]$hdfs dfs - ls - R   /result
drwxr - xr - x   - harli supergroup          0 2015 - 05 - 31 10:13 /result/appstat1
- rw - r - - r - -   1harli supergroup          0 2015 - 05 - 31 10:13 /result/appstat1/_SUCCESS
- rw - r - - r - -   1harli supergroup         23 2015 - 05 - 31 10:13 /result/appstat1/part - 00000

[harli@ cluster04 cluster]$hdfs dfs - text   /result/appstat1/part - 00000
(qq,10)
(360,6)
(yy,4)
```

三、移动互联网上的 DAU 及 MAU 的统计

对动互联网上的 DAU 及 MAU 的统计时，需要注意对用户的去重处理：每个用户由字段“IMEI”唯一标识，统计时需要去除重复用户。

统计移动互联网上的 DAU 的代码如下：

```
scala > mobile. map(x => (x(bcfields. value. indexOf("IMEI")) + ":" + x(bcfields. value. indexOf("
Time")))). distinct(). map(x => (x. split(":")(1),1)). reduceByKey(_ + _). sortByKey(). collect
res53:Array[(String,Int)] = Array((2015 - 05 - 01,2),(2015 - 05 - 02,2),(2015 - 05 - 03,2),
(2015 - 05 - 04,2),(2015 - 05 - 05,2),(2015 - 06 - 01,1),(2015 - 06 - 02,2),(2015 - 06 - 03,
2),(2015 - 06 - 04,2),(2015 - 06 - 05,2),(2015 - 06 - 10,1))
```

首先，为了统计 DAU 时对用户进行去重，通过第一个 map 操作，将唯一标识用户信息的“IMEI”字段和标识日访问的时间“Time”字段进行合并；然后调用 distinct()方法进行去重；最后从合并数据中提取出“Time”字段，并进行计数统计。

同样，为了方便查看，这里通过 collect 在界面上显示结果，在实际的应用中，可以保存到文件中，具体参考对移动互联网数据的不同应用使用情况的统计中的保存文件代码部分。后续代码中都会通过 collect 方式显示统计结果（collect 适合在调试代码或查看少量数据时使用，大量数据处理结果的查看，可以通过 take 等方法）。

统计移动互联网上的 MAU 的代码如下：

```
scala > mobile. map{ x =>
     |   val t = x(bcfields. value. indexOf("Time"))
     |   val m = t. substring(0,t. lastIndexOf(" - "))
```

```
     |   x(bcfields.value.indexOf("IMEI")) + ":" + m
     | }.distinct().map(x => (x.split(":")(1),1)).reduceByKey(_ + _).sortByKey().collect
15/05/31 10:52:06 INFOFileInputFormat:Total input paths to process:1
res7:Array[(String,Int)] = Array((2015 - 05,10),(2015 - 06,10))
```

MAU 的统计和 DAU 的统计仅仅在统计的时间上存在差异，这里首先在合并“IMEI”字段和时间“Time”字段时，将时间“Time”字段转换为月份，之后的处理和 DAU 的统计是一样的。

四、在不同应用中的上下行流量统计

对移动互联网数据的不同应用的流量统计可以得到不同应用的基础分析信息，比如可以结合各个应用的使用次数，各个应用的类型（比如视频应用、网页浏览应用等）等信息分析各个应用在流量使用上是否合理，以及分析出不同应用对网速的不同要求等。

具体上下行流量的统计代码如下：

```
scala > mobile.map { x =>
     |   val ub = x(bcfields.value.indexOf("UplinkBytes")).toDouble
     |   val db = x(bcfields.value.indexOf("DownlinkBytes")).toDouble
     |  (x(bcfields.value.indexOf("APP")),List[Double](ub,db))
     | }.reduceByKey((x,y) => List(x(0) + y(0),x(1) + y(1))).collect
res9:Array[(String,List[Double])] = Array((yy,List(34.0,3479.0)),(qq,List(117.0,6195.0)),
(360,List(54.0,2714.0)))
```

首先，通过 map 方法，将每条访问记录转换为（应用名称，［上行流量，下行流量］）的数据类型；然后通过 reduceByKey 方法对各个应用的上下行流量进行统计分析。

需要说明的是，实际情况下各种应用中会包含不同的子应用（即一个大类型的应用中会有子类型的应用），比如在 QQ 应用中，会包含 QQ 空间、文本聊天、语音聊天以及视频聊天等。在真实的移动互联网数据中，通常会使用两个字段，即大业务类型和小业务类型来表示 QQ 应用及其具体的子应用，这时候可以针对大应用和小应用进行统计以实现更精确的分析。

对移动互联网数据进行统计分析之后，可以进一步通过可视化工具进行呈现。下面应用程序统计如图 2.74 所示。

腾讯、阿里、百度及360占据TOP20移动应用的绝大多数

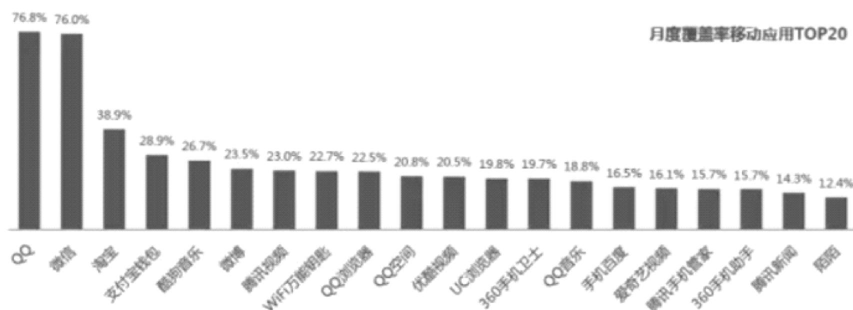


图 2.74　移动互联网 Top20 应用的占比图

通过统计分析并以图表方式呈现后，可以让用户更易于了解移动互联网的行业发展情况。



Section 2.6 Spark RDD 实践中的常见问题与解答

在 Spark RDD 的实践中，初学者经常会碰到一些问题，比如如何设置加载文件后构建的 RDD 的分区数，如何加载各种文件系统中的文件等。这里列出了出现频率比较高的问题，在前面章节的实践案例与解析基础上，给出问题的解答。

1. 如何控制加载的文件系统，即如何设置文件加载时的 scheme

Spark 集群构建在本地文件系统时：默认使用的是本地文件系统，对应于 scheme 为 file://，即/home/path 实际对应 file:///home/path。

Spark 集群构建在 HDFS 文件系统上时：默认使用的是 Hadoop 配置的 defaultFS 文件系统，一般为 HDFS，因此，scheme 默认为 hdfs://，即/user/path 实际对应 hdfs://defaultFS/user/path。如果是构建在本地文件系统的话，默认使用的就是本地文件系统了，也就是对应 file://。

其中 defaultFS 为 HDFS 中配置的文件系统信息，一般设置为 master:8020。

2. 应用程序中的 println 信息没有输出到界面上

分布式计算时，对应在 Executor 上执行的 Task 的输出也同样位于 Executor 所在 Worker 机器节点上，这也是为什么不能在 Driver 的控制台上看到 println 等对应的输出信息的原因。可以通过页面上对应的 Executor 上的 stdout/stderr 两个输出文件进行查询。可以参考章节 2.4.3Spark 提交应用的调试实例部分。在后面的实例中，会给出不同集群模式和部署模式下，日志信息如何查找的案例，包括 Standalone 和 Yarn 两种集群下的 Executor 的日志查询和 Driver 的日志查询。

3. cache 为什么没有起作用，或没有存储到磁盘上

参见章节 2.2.7RDD 的持久化案例与解析部分，cache 对应的存储级别及其含义如下：

存储级别（Storage Level）	含　义
MEMORY_ONLY	将 RDD 以反序列化（deserialized）的 Java 对象存储到 JVM。如果 RDD 不能被内存装载，一些分区将不会被缓存，并且在需要的时候被重新计算。这是默认的级别

MEMORY_ONLY 缓存级别的缓存，只有在内存能装载下时，RDD 的分区数据才存储在内存中，否则会被丢弃，在需要时重新计算。如果选择 MEMORY_ONLY 级别，内存装载不下时 RDD 的分区数据是不会持久化到磁盘的，所以要根据所选择的存储级别去分析实际的存储情况。

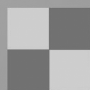
另外，只有在触发的情况下，才会计算出 RDD，然后进行缓存，如果只计算了 RDD 中的部分分区（比如用 take 等进行触发的话），也就只有计算后的这部分分区会缓存，而不是缓存整个 RDD。

4. 如何判断一个 RDD 的算子操作有 Shuffle 过程

判断一个 RDD 的算子是否有 Shuffle 过程，并不是依据该 RDD 是否为 ShuffledRDD，而是根据算子操作得到的结果 RDD，查询其父依赖关系中是否包含了 ShuffleDependency 类型的依赖，如果包含，则操作过程需要进行 Shuffle。比如 CoGroupedRDD 类，可以自己查看下它的 getDependencies 方法的源码，将会看到，CoGroupedRDD 的父依赖有可能为 ShuffleDependency 的，所以 CoGroupedRDD 有可能需要 Shuffle 过程。

第 3 章　Spark SQL实践案例与解析

3.1　Spark SQL 概述
3.2　DataFrame 处理的案例与解析
3.3　Spark SQL 处理各种数据源的案例与解析
3.4　基于 Hive 的人力资源系统数据处理案例与解析



Section 3.1 Spark SQL 概述

Spark SQL 在伯克利数据分析协议栈中的位置如图 3.1 所示：

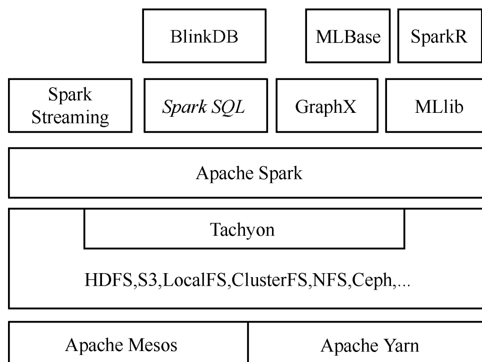


图 3.1　伯克利数据分析协议栈

Spark SQL 在 Spark 内核基础上提供了对结构化数据的处理，在 Spark1.3 版本中，Spark SQL 不仅可以作为分布式的 SQL 查询引擎，还引入了新的 DataFrame 编程模型。

在 Spark1.3 版本中，Spark SQL 不再是 Alpha 版本，除了提供更好的 SQL 标准兼容之外，还引进了新的组件 DataFrame。同时，Spark SQL 数据源 API 也实现了与新组件 DataFrame 的交互，允许用户直接通过 Hive 表、Parquet 文件以及一些其他数据源生成 DataFrame。用户可以在同一个数据集上混合使用 SQL 和 DataFrame 操作符。新版本还提供了从 JDBC 读写表的能力，可以更原生地支持 Postgres、MySQL 及其他 RDBMS 系统。

Spark SQL 所有功能的入口点是 SQLContext，或它的一个子类。只需要一个 SparkContext 实例就可以构建一个基本的 SQLContext。

再次强调，spark - shell 除了帮我们构建了 SQLContext 实例外，还帮我们导入了隐式转换：import sqlContext. implicits. _。在以 spark - submit 方式提交的应用程序中，需要手动导入该隐式转换才能访问某些 API。

Section 3.2 DataFrame 处理的案例与解析

Spark 内核的 RDD API 通过函数式编程的模式把分布式数据处理转换成分布式数据集（distributed collections）的处理，提供了更高抽象层次的 API。原本用 Hadoop MapReduce 实现的代码需要上千行，而在 Spark 这个 API 上可以减少到数十行。

随着 Spark 的不断壮大，为了向更广泛的受众群体提供分布式处理，Spark 1.3 引入了新的 DataFrame 编程模型，在概念上，DataFrame 类似于关系数据库的表，或在 R/Python 中的 DataFrame，这一组件的引入，简化了 Spark SQL 的处理，极大方便了数据科学方面的应用。

DataFrame 编程模型极大地简化了 Spark SQL 的编程复杂度，在开始 DataFrame 的实践案例与解析之前，需要对 DataFrame 编程模型有一个基础了解。

3.2.1 DataFrame 编程模型

Spark SQL 允许 Spark 执行用 SQL 语言、HiveQL 语言或者 Scala 语言表示的关系查询。在 Spark 1.3 之前，这个模块的核心是 SchemaRDD 类型。SchemaRDD 由行（Row）对象组成，行对象通过 scheme 来描述行中每一列的数据类型。

而在 Spark 1.3 中，引入了 DataFrame 来重命名 SchemaRDD 类型，在 Spark1.3 中，DataFrame 是一个以命名列方式组织的分布式数据集，在概念上类似于关系型数据库中的一个表，也相当于 R/Python 中的 Data Frames。DataFrame 可以由结构化数据文件转换得到，或从 Hive 中的表得来，也可以转换自外部数据库或现有的 RDD。

DataFrame 编程模型具有的功能特性有：

1）从 KB 到 PB 级的数据量支持。

2）多种数据格式和多种存储系统支持。

3）通过 Spark SQL 的 Catalyst 优化器进行先进的优化，生成代码。

4）为 Python、Java、Scala 和 R 语言（SparkR）提供 API。

注意： 目前 DataFrame API 支持 Scala、Java 以及 Python。以下章节的实践基于 Scala 语言。

3.2.2 DataFrame 基本操作案例与解析

这一节给出了一个集团公司对人事信息处理场景的简单案例，详细分析 DataFrame 上的各种常用操作，包括集团子公司间的职工人事信息的合并、职工的部门相关信息查询、职工信息的统计、关联职工与部门信息的统计，以及如何将各种统计得到的结果存储到外部存储系统等。

在案例中，涉及的 DtaFrame 实例内容包括从外部文件构建 DataFrame，在 DataFrame 上比较常用的操作，多个 DataFrame 之间的操作，以及 DataFrame 的持久化操作等内容。

一、数据准备

数据准备包含两部分内容，一是实践案例的数据设计部分，二是将数据文件上传到 HDFS 存储系统上。

1. 创建本地文件的目录

```
hadoop-2.6.0   resources   spark-1.3.0-bin-hadoop2.4   tmp
[harli@wxx215 cluster_13]$mkdir test
[harli@wxx215 cluster_13]$ls
hadoop-2.6.0   resources   spark-1.3.0-bin-hadoop2.4   test   tmp
[harli@wxx215 cluster_13]$cd test/
```

在本地文件目录中构建包含实践数据的文件，包含员工信息的文件、新增员工信息的文件以及部门信息的文件。

2. 编辑文件 people. json

```
[harli@ wxx215 test]$vim people. json
{"name":"Michael","job number":"001","age":33,"gender":"male","deptId":1,"salary":3000}
{"name":"Andy","job number":"002","age":30,"gender":"female","deptId":2,"salary":4000}
{"name":"Justin","job number":"003","age":19,"gender":"male","deptId":3,"salary":5000}
{"name":"John","job number":"004","age":32,"gender":"male","deptId":1,"salary":6000}
{"name":"Herry","job number":"005","age":20,"gender":"female","deptId":2,"salary":7000}
{"name":"Jack","job number":"006","age":26,"gender":"male","deptId":3,"salary":3000}
```

people. json 文件包含了员工的相关信息，每一列分别对应：员工姓名、工号、年龄、性别、部门 ID 以及薪资。

3. 编辑文件 newPeople. json

```
[harli@ wxx215 test]$vim newPeople. json
{"name":"John","job number":"007","age":32,"gender":"male","deptId":1,"salary":4000}
{"name":"Herry", "job number":"008","age":20,"gender":"female","deptId":2,"salary":5000}
{"name":"Jack","job number":"009","age":26,"gender":"male","deptId":3,"salary":6000}
```

该文件对应新入职员工的信息，这里简化了不同子集团员工信息的结构差异，如果子集团员工具有不同的信息结构，可以通过在样本类 people 中将缺失的列设置为默认值，从而保证所有的员工信息在结构上的一致性。

4. 编辑文件 department. json

```
[harli@ wxx215 test]$vim department. json
{"name":"Development Dept",    "deptId":1}
{"name":"Personnel Dept",      "deptId":2}
{"name":"Testing Department","deptId":3}
```

department. json 文件是员工的部门信息，内容包含部门的名称和部门 ID。其中，部门 ID 对应员工信息中的部门 ID，即员工的 deptId 列。

5. 上传文件

```
[harli@ wxx215 cluster_13]$cd hadoop-2. 6. 0/
[harli@ wxx215 hadoop-2. 6. 0]$. /bin/hdfs dfs -put . . /test /user/harli
15/04/03 10:48:32 WARN util. NativeCodeLoader:Unable to load native-hadoop library for your platform. . .  using builtin-java classes where applicable
[harli@ wxx215 hadoop-2. 6. 0]$. /bin/hdfs dfs -ls /user/harli
15/04/03 10:48:50 WARN util. NativeCodeLoader:Unable to load native-hadoop library for your platform. . .  using builtin-java classes where applicable
Found 5 items
drwxr-xr-x   - harli supergroup          0 2015-03-31 16:41 /user/harli/json
drwxr-xr-x   - harli supergroup          0 2015-03-31 16:44 /user/harli/parquet
drwxr-xr-x   - harli supergroup          0 2015-04-01 14:09 /user/harli/resources
drwxr-xr-x   - harli supergroup          0 2015-03-31 16:10 /user/harli/save. json
drwxr-xr-x   - harli supergroup          0 2015-04-03 10:48 /user/harli/test
```

将所需的三个文件上传到 Hadoop 集群中，作为实践案例的集群数据。

1）查看上传结果：

```
[harli@ wxx215 hadoop-2. 6. 0]$. /bin/hdfs dfs -ls /user/harli/test
```

```
15/04/03 10:54:59 WARN util.NativeCodeLoader:Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 3 items
-rw-r--r--   3harli supergroup   126 2015-04-03 10:48 /user/harli/test/department.json
-rw-r--r--   3harli supergroup   282 2015-04-03 10:48 /user/harli/test/newPeople.json
-rw-r--r--   3harli supergroup   564 2015-04-03 10:48 /user/harli/test/people.json
```

通过 Hadoop 的命令行，查看上传文件是否成功。可以看到，三个文件已经成功上传到 HDFS 存储系统上。

2）查看界面显示：Web Interface（http://namenode:50070/explorer.html#/user/harli/test）的界面信息如图 3.2 所示。

Permission	Owner	Group	Size	Replication	Block Size	Name
-rw-r--r--	harli	supergroup	126 B	3	128 MB	department.json
-rw-r--r--	harli	supergroup	282 B	3	128 MB	newPeople.json
-rw-r--r--	harli	supergroup	564 B	3	128 MB	people.json

图 3.2　Hadoop 文件系统上传结果的界面

登录 HDFS 的 namenode 节点（namenode 节点为启动 NameNode 进程的节点，当前环境下的节点地址为 192.168.70.214。）的 50070 端口，在 Utilities 菜单上单击 Browse the file system 命令，开始浏览当前 HDFS 的文件系统信息，内容包含文件的访问权限、所有者、所有者所在组、文件大小、复制因子、块大小以及文件的名字。

可以在路径导航栏部分输入指定的目录，然后单击“Go!”按钮跳转到该目录下。在浏览目录文件时，可以通过单击文件名来打开文件具体信息的窗口，在打开的窗口上还提供了文件下载的功能。

二、启动交互式界面

当前以集群模式启动 spark-shell 应用，在 spark 部署目录下，输入以下命令：

```
[harli@wxx215 spark-1.3.0-bin-hadoop2.4]$./bin/spark-shell --master
spark://192.168.70.214:7077
```

启动后出现如下信息：

```
[harli@wxx215 spark-1.3.0-bin-hadoop2.4]$./bin/spark-shell --master
spark://192.168.70.214:7077
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
15/04/03 10:58:54 WARN util.NativeCodeLoader:Unable to load native-hadoop library for your plat-
```

```
form... using builtin - java classes where applicable
15/04/03 10:58:54 INFO spark.SecurityManager:Changing view acls to:harli
15/04/03 10:58:54 INFO spark.SecurityManager:Changing modify acls to:harli
15/04/03 10:58:54 INFO spark.SecurityManager:SecurityManager:authentication disabled;ui acls disa-
bled;users with view permissions:Set(harli);users with modify permissions:Set(harli)
15/04/03 10:58:54 INFO spark.HttpServer:Starting HTTP Server
15/04/03 10:58:55 INFO server.Server:jetty - 8.y.z - SNAPSHOT
15/04/03 10:58:55 INFO server.AbstractConnector:Started SocketConnector@0.0.0.0:57536
15/04/03 10:58:55 INFO util.Utils:Successfully started service' HTTP class server' on port 57536.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Scala version 2.10.4(Java HotSpot(TM)Server VM,Java 1.7.0)
Type in expressions to have them evaluated.
Type:help for more information.
```

在启动界面最后会生成 SparkContext 和 SQLContext 两个实例，对应 sc 和 sqlContext。直接使用 sqlContext 进行操作：

```
15/04/03 10:59:03 INFOrepl.SparkILoop:Created spark context..
Spark context available as sc.
15/04/03 10:59:03 INFOrepl.SparkILoop:Created sql context(with Hive support)..
SQL context available assqlContext.
15/04/03 10:59:05 INFO cluster.SparkDeploySchedulerBackend:Registered executor:Actor[akka.tcp:
//sparkExecutor@wxx225:18129/user/Executor# - 1906560325] with ID 0
15/04/03 10:59:05 INFO cluster.SparkDeploySchedulerBackend:Registered executor:Actor[akka.tcp:
//sparkExecutor@wxx223:52922/user/Executor# - 1906560325] with ID 2
15/04/03 10:59:05 INFO storage.BlockManagerMasterActor:Registering block managerwxx225:18869
with 265.0 MB RAM,BlockManagerId(0,wxx225,18869)
15/04/03 10:59:05 INFO storage.BlockManagerMasterActor:Registering block managerwxx223:44620
with 265.0 MB RAM,BlockManagerId(2,wxx223,44620)
```

按【Enter】键后进入交互式界面，开始案例的操作。

从上述信息中，我们可以看到交互式工具 spark - shell 启动后创建的是带 Hive 支持的 SQLContext，对应的提示信息为：repl.SparkILoop：Created sql context （with Hive support）。

查找对应的源码，我们可以看到，在 repl.SparkLoop 类中构建 sqlContext 时使用的是 HiveContext 类，源码如下：

```
@DeveloperApi
defcreateSQLContext():SQLContext = {
    val name = "org.apache.spark.sql.hive.HiveContext"
    val loader = Utils.getContextOrSparkClassLoader
    try {
sqlContext = loader.loadClass(name).getConstructor(classOf[SparkContext])
              .newInstance(sparkContext).asInstanceOf[SQLContext]
```

```
logInfo("Created sql context(with Hive support)..")
      } catch {
            case cnf:java.lang.ClassNotFoundException =>
sqlContext = new SQLContext(sparkContext)
logInfo("Created sql context..")
      }
      sqlContext
}
```



当Spark带Hive编译时，对应创建的就是HiveContext实例，而当该实例构建失败时，创建的是SQLContxt实例。

注意：当我们使用spark-submit来提交应用程序时，在应用程序中，应该用相同的方式去构建SQLContext实例，通过该实例进行Spark SQL的操作。同时，使用spark-shell时，已经自动导入一些隐式转换，对应的，使用spark-submit提交时，应在代码中手动加入，如import sqlContext.implicits._。

三、案例实操

这部分内容对员工信息以及员工的部门信息进行处理，以下是各类数据操作的具体操作及分析。

1. 修改日志等级

```
import org.apache.log4j.{Level,Logger}
Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
```

这个操作的目的是为了简化界面的输出。在交互界面上输入以上代码，将日志级别调整到Level.WARN。

2. 加载文件

```
scala > val people = sqlContext.jsonFile("hdfs://wxx214:9000/user/harli/test/people.json")
15/04/03 11:03:01 INFOmapred.FileInputFormat:Total input paths to process:1
people:org.apache.spark.sql.DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
scala > val dept    = sqlContext.load("hdfs://wxx214:9000/user/harli/test/department.json","json")
15/04/03 11:03:01 INFOmapred.FileInputFormat:Total input paths to process:1
dept:org.apache.spark.sql.DataFrame = [deptId:bigint,name:string]
```

这里提供了两种方式加载之前上传到HDFS存储系统上的员工信息文件和部门信息文件，可以看到，文件加载后得到了两个DataFrame实例：people和dept，同时根据文件内容自动地推导出两个DataFrame实例的schema信息，schema信息包含了列的名字以及对应的数据类型，如dept的schema信息为[deptId:bigint,name:string]。

构建DataFrame的其他方式及其解析参见后续章节的内容。

3. 以表格形式查看DataFrame信息

```
scala > people.show
15/04/03 11:05:06 INFOmapred.FileInputFormat:Total input paths to process:1
agedeptId gender job number name      salary
33   1      male    001        Michael  3000
30   2      female  002        Andy     4000
19   3      male    003        Justin   5000
```

```
32  1  male    004  John    6000
20  2  female  005  Herry   7000
26  3  male    006  Jack    3000
scala > people.show
15/04/03 11:05:55 INFOmapred.FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
33  1  male    001  Michael 3000
30  2  female  002  Andy    4000
19  3  male    003  Justin  5000
32  1  male    004  John    6000
20  2  female  005  Herry   7000
26  3  male    006  Jack    3000
```

通过 show 方法，可以以表格形式输出各个 DataFrame 的内容。如上所示，可以看到该 DataFrame 加载的文件路径数，以及包含的各个列名和数据内容，默认情况下会显示 DataFrame 的前 20 条记录，可以通过设置 show 方法的参数来指定输出的记录条数，如 people.show(10)，显示前 10 条记录。

4. DataFrame 基本信息的查询

```
scala > people.columns
res22: Array[String] = Array(age,deptId,gender,job number,name,salary)
scala > people.count
15/04/03 11:11:14 INFOmapred.FileInputFormat:Total input paths to process:1
res23: Long = 6
scala > people.take(3)
15/04/03 11:11:15 INFOmapred.FileInputFormat:Total input paths to process:1
res24: Array[org.apache.spark.sql.Row] = Array([33,1,male,001,Michael,3000],[30,2,female,
002,Andy,4000],[19,3,male,003,Justin,5000])
scala > people.toJSON.collect
res25:Array[String] = Array({"age":33,"deptId":1,"gender":"male","job number":"001","
name":"Michael","salary":3000},{"age":30,"deptId":2,"gender":"female","job number":"
002","name":"Andy","salary":4000},{"age":19,"deptId":3,"gender":"male","job number":"
003","name":"Justin","salary":5000},{"age":32,"deptId":1,"gender":"male","job num-
ber":"004","name":"John","salary":6000},{"age":20,"deptId":2,"gender":"female","job
number":"005","name":"Herry","salary":7000},{"age":26,"deptId":3,"gender":"male","job
number":"006","name":"Jack","salary":3000})
```

这部分内容针对员工信息的 DataFrame，即 people，进行一些基本信息的查询操作，具体包含：

1）使用 DataFrame 的 columns 方法，查询 people 包含的全部列（Columns）信息，以数组形式返回列名组。

2）使用 DataFrame 的 count 方法，统计 people 包含的记录条数，即员工个数。

3）使用 DataFrame 的 take 方法，获取前三条员工记录信息，并以数组形式呈现出来。

4）最后使用 DataFrame 的 toJSON 方法，将 people 转换为 JsonRDD 类型，并使用 RDD 的 collect 方法返回其包含的员工信息。



5. 对员工信息进行条件查询，并输出结果

```
scala > people. filter( "gender = male " ). count
15/04/03 11:08:13 INFOmapred. FileInputFormat:Total input p,aths to process:1
res12:Long = 4
scala > people. filter($"gender"  !== "female" ). count
15/04/03 11:08:14 INFOmapred. FileInputFormat:Total input paths to process:1
res13:Long = 4
scala > people. filter($"age"  > 25). show
15/04/03 11:08:15 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary
33   1        male    001          Michael   3000
30   2        female  002          Andy      4000
32   1        male    004          John      6000
26   3        male    006          Jack      3000
scala > people. where($"age"  > 25). show
15/04/03 11:08:17 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary
33   1        male    001          Michael   3000
30   2        female  002          Andy      4000
32   1        male    004          John      6000
26   3        male    006          Jack      3000
scala > people. where($"age"  > 25 && $"gender" === "male" ). show
15/04/03 17:01:31 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary
33   1        male    001          Michael   3000
32   1        male    004          John      6000
26   3        male    006          Jack      3000
scala > people. where( 'age > 25). show
15/04/03 11:08:35 INFO mapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary
33   1        male    001          Michael   3000
30   2        female  002          Andy      4000
32   1        male    004          John      6000
26   3        male    006          Jack      3000
```

这部分内容针对员工信息的 DataFrame，即 people，进行一些条件查询的操作，具体包含：

1）使用 count 方法统计了“gender”列为“male”的员工数量。

2）基于“age”和“gender”这两列，使用不同的查询条件，不同的 DataFrame API，即 where 和 filter 方法，对员工信息进行过滤。

3）最后仍然使用 show 方法，将查询结果以表格形式呈现出来。

4）在各个例子中，使用了几种不同的方式，作为查询条件的参数。

6. 根据指定的列名，以不同方式进行排序

```
scala > people. sort($"job number". asc,col( "deptId" ). desc). show
15/04/03 11:12:49 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary
33   1        male    001          Michael   3000
30   2        female  002          Andy      4000
```

```
19   3    male    003    Justin   5000
32   1    male    004    John     6000
20   2    female  005    Herry    7000
26   3    male    006    Jack     3000
scala > people. sort($"job number"). show(3)
15/04/03 11:13:35 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
33   1    male    001    Michael  3000
30   2    female  002    Andy     4000
19   3    male    003    Justin   5000
scala > people. sort("job number"). show(3)
15/04/03 11:13:35 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
33   1    male    001    Michael  3000
30   2    female  002    Andy     4000
19   3    male    003    Justin   5000
scala > people. sort($"job number". asc). show
15/04/03 11:13:57 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
33   1    male    001    Michael  3000
30   2    female  002    Andy     4000
19   3    male    003    Justin   5000
32   1    male    004    John     6000
20   2    female  005    Herry    7000
26   3    male    006    Jack     3000
```

这部分内容针对员工信息的 DataFrame，即 people，基于“job number”和“deptId”列，用 sort 方法，并以不同方式进行排序，并输出结果，具体包含：

1）以先“job number”列升序，然后再“deptId”列降序的方式，对 people 进行排序，并输出排序后的内容；这里给出了两种指定列的方式。

2）以“job number”列进行默认排序（升序），并显示排序后的 3 条记录；这里也给出了两种指定列的方式。

3）以“job number”列，指定降序方式进行排序，并显示排序后的 3 条记录。

7. 为员工信息增加一列：等级（“level”）

```
scala > people. withColumn("level",people("age")/ 10). show
15/04/03 12:16:19 INFOmapred. FileInputFormat:Total i;
nput paths to process:1
agedeptId gender job number name    salary level
33   1    male    001    Michael  3000   3. 3
30   2    female  002    Andy     4000   3. 0
19   3    male    003    Justin   5000   1. 9
32   1    male    004    John     6000   3. 2
20   2    female  005    Herry    7000   2. 0
26   3    male    006    Jack     3000   2. 6
```

这部分内容针对员工信息的 DataFrame，即 people，通过 withColumns 方法增加了新的一列等级信息，列名为“level”。

其中，withColumns 方法的“level”参数指定了新增列的列名，第二个参数指定了该列的实例，即通过“age”列转换得到的新列；而 people("age")方法则调用了 DataFrame 的 apply 方法，返回“age”列名对应的列。



8. 修改工号列名

```
scala > people. columns
res85:Array[String] = Array(age,deptId,gender,job number,name,salary)
scala > people. withColumnRenamed("job number","jobId"). columns
res86:Array[String] = Array(age,deptId,gender,jobId,name,salary)
```

这部分内容针对员工信息的 DataFrame，即 people，通过 withColumnRenamed 方法修改其现有的列名。

在示例中，将 people 的“job number”列名修改为“jobId”。通过交互式反馈信息可以看到列名已经被修改。

注意，修改的列名必须存在，如果不存在，不会报错，但列名不会修改，例如：

```
scala > val rnDept = people. withColumnRenamed("job numbe","jobId")
rnDept:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
scala > rnDept. columns
res150:Array[String] = Array(age,deptId,gender,job number,name,salary)
scala > people. columns
res151:Array[String] = Array(age,deptId,gender,job number,name,salary)
```

在示例中，指定修改的“job numbe”列名拼写错误，可以看到列名并没有被修改。

9. 增加新员工

```
scala > val newPeople = sqlContext. jsonFile("hdfs://wxx214:9000/user/harli/test/newPeople. json")
15/04/03 12:18:30 INFOmapred. FileInputFormat:Total input paths to process:1
newPeople:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:
string,name:string,salary:bigint]
scala > newPeople. show
15/04/03 12:19:16 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name   salary
32  1     male    007      John    4000
20  2     female  008      Herry   5000
26  3     male    009      Jack    6000
scala > people. unionAll(newPeople). show
15/04/03 12:18:32 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 12:18:32 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name      salary
33  1     male    001      Michael  3000
30  2     female  002      Andy     4000
19  3     male    003      Justin   5000
32  1     male    004      John     6000
20  2     female  005      Herry    7000
26  3     male    006      Jack     3000
32  1     male    007      John     4000
20  2     female  008      Herry    5000
```

```
26   3        male    009          Jack   6000
```

示例中使用 jsonFile 方法加载了新员工信息的文件，然后调用 people 的 unionAll 方法，将新加载的 newPeople 合并进来。

可以看到，最终合并时，对应的输入文件路径为 2，即对应了新旧两个员工信息的文件，这是因为加载文件是 lazy 性质的。这里由于没有对 DataFrame 进行缓存，因此合并时会重新进行加载。

10. 查同名员工

```
scala > val groupName = people. unionAll( newPeople). groupBy( col( "name" ) ). count
groupName:org. apache. spark. sql. DataFrame = [ name:string,count:bigint]
scala > groupName. show
15/04/03 13:44:10 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 13:44:10 INFOmapred. FileInputFormat:Total input paths to process:1
name      count
Justin    1
Jack      2
John      2
Andy      1
Michael   1
Herry     2
scala > groupName. filter($"count"  >1). show
15/04/03 13:44:12 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 13:44:12 INFOmapred. FileInputFormat:Total input paths to process:1
name   count
Jack   2
John   2
Herry  2
scala > people. unionAll( newPeople). groupBy( col( "name" ) ). count. filter($"count"  >1). show
15/04/03 13:44:13 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 13:44:13 INFOmapred. FileInputFormat:Total input paths to process:1
name   count
Jack   2
John   2
Herry  2
```

示例中，首先通过 unionAll 方法将 people 和 newPeople 进行合并，然后使用 groupBy 方法将合并后的 DataFrame 按照“name”列进行分组，分组操作会得到一个 GroupData 类的实例 groupName，实例会自动带上分组的列，以及“count”列；GroupData 类型提供了一组非常有用的统计操作，这里继续调用它的 count 方法，最终实现对员工名字的分组计数。

接着上对 groupName 实例进行过滤操作，使用 filter 方法，获取“name”列的计数大于 1 的内容，并表格形式予以呈现。

最后一个示例，使用函数式编程范式对前两个的合并，得到的结果是一样的。

11. 分组统计信息

```
scala > val depAgg = people. groupBy( "deptId" ). agg( Map(
     |    "age" - > "max",
     |    "gender" - > "count"
```



```
    |))
depAgg:org.apache.spark.sql.DataFrame = [deptId:bigint,MAX(age#0L):bigint,COUNT(gender#2):
bigint]
scala > depAgg.show
15/04/03 15:04:56 INFOmapred.FileInputFormat:Total input paths to process:1
deptId MAX(age#0L)COUNT(gender#2)
1       33          2
2       30          2
3       26          2
scala > depAgg.toDF("deptId","maxAge","countGender").show
15/04/03 15:04:57 INFOmapred.FileInputFormat:Total input paths to process:1
deptIdmaxAgecountGender
1       33    2
2       30    2
3       26    2
```

这里继续对分组统计实例进行解析。首先针对 people 的“deptId”列进行分组，分组后得到的 GroupData 实例继续调用 agg 方法，分别对“age”列求最大值，对“gender”进行计数。

由交互式界面反馈可知，最终返回 DataFrame，并且它的 schema 为[deptId:bigint,MAX(age#0L):bigint,COUNT(gender#2):bigint]，即除了带上分组用的“deptId”列外，还带上列聚合操作后的两列信息。

最后一步，调用 DataFrame 的 toDF 方法，重新命名之前聚合得到的 depAgg 的全部列名，增加了列名的可读性。

12. 名字去重

```
scala > val unionPeople = people.unionAll(newPeople).select("name").show
15/04/03 15:17:23 INFOmapred.FileInputFormat:Total input paths to process:1
15/04/03 15:17:23 INFOmapred.FileInputFormat:Total input paths to process:1
name
Michael
Andy
Justin
John
Herry
Jack
John
Herry
Jack
unionPeople:Unit = ()
scala > val unionPeople = people.unionAll(newPeople).select("name").distinct.show
15/04/03 15:17:35 INFOmapred.FileInputFormat:Total input paths to process:1
15/04/03 15:17:35 INFOmapred.FileInputFormat:Total input paths to process:1
name
Justin
Jack
John
Andy
```

```
Michael
Herry
unionPeople:Unit = ()
```

示例中首先显示新旧员工信息合并后的“name”列，作为后续去重的比较对象。

通过 unionAll 新旧员工信息，并只选择其中的“name”列信息后，出现的“name”信息就出现列重复，通过继续调用 DataFrame 的 distinct 方法后，可以去除重复的记录数据。

13. 对比新旧员工表

```
scala > people.select("name").except(newPeople.select($"name")).show
15/04/03 15:20:08 INFOmapred.FileInputFormat:Total input paths to process:1
15/04/03 15:20:08 INFOmapred.FileInputFormat:Total input paths to process:1
name
Michael
Justin
Andy
scala > people.select("name").intersect(newPeople.select($"name")).show
15/04/03 15:20:11 INFOmapred.FileInputFormat:Total input paths to process:1
15/04/03 15:20:11 INFOmapred.FileInputFormat:Total input paths to process:1
name
Jack
John
Herry
```

示例中，包含了对 people 和 newPeople 两个员工信息的“name”列的两种比较方式，具体如下。

1）第一种：分别选取 people 和 newPeople 两个员工信息的“name”列，然后通过调用 except 方法，获取在 people 中出现、但同时不在 newPeople 中出现的“name”信息，最后以表格形式呈现结果。

2）第二种：求“name”的交集，即分别选取 people 和 newPeople 两个员工信息的“name”列，然后通过调用 intersect 方法，获取在 people 中出现、但同时又在 newPeople 中出现的“name”信息，最后以表格形式呈现结果。

14. join 两个 DataFrame 实例

```
scala > people.join(dept,people("deptId") === dept("deptId"),"outer").show
15/04/03 15:25:55 INFOmapred.;FileInputFormat:Total input paths to process:1
15/04/03 15:25:55 INFOmapred.FileInputFormat:Total input paths to process:1
agedeptId gender job number name     salary deptId name
33  1     male   001        Michael  3000   1      Development Dept
32  1     male   004        John     6000   1      Development Dept
20  2     female 005        Herry    7000   2      Personnel Dept
30  2     female 002        Andy     4000   2      Personnel Dept
19  3     male   003        Justin   5000   3      Testing Department
26  3     male   006        Jack     3000   3      Testing Department
```

在示例中，people 通过调用 join 方法，基于 people 的“deptId”列与 dept 的“deptId”列进行 outer join 联合操作。由于 people 与 dept 的两个 DataFrame 中用于联合的列名相同，都是“dept”，因此，指定联合条件表达式时，需要指出列所属的具体 DataFrame 实例，否则

报错。

在作为 join 联合条件的列表达式中，如果列名不同，则可以直接使用列名，比如：



```
scala > val rnDept = dept. withColumnRenamed( "deptId" ," id" )
rnDept:org. apache. spark. sql. DataFrame = [ id:bigint,name:string]
scala > val jionP = people. join( rnDept,$" deptId" ===$" id" ," outer" )
jionP:org. apache. spark. sql. DataFrame = [ age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint,id:bigint,name:string]
scala > jionP. show
15/04/03 15:42:19 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 15:42:19 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary id name
33  1       male    001        Michael  3000  1  Development Dept
32  1       male    004        John     6000  1  Development Dept
30  2       female  002        Andy     4000  2  Personnel Dept
20  2       female  005        Herry    7000  2  Personnel Dept
26  3       male    006        Jack     3000  3  Testing Department
19  3       male    003        Justin   5000  3  Testing Department
```

15. join 后按部门名分组统计

```
scala > valjoinGp = jionP. groupBy( dept( " name" ) ). agg( Map(
      |    " age" - > " max" ,
      |    " gender" - > " count"
      | ) )
joinGp:org. apache. spark. sql. DataFrame = [ name:string,MAX( age#0L) :bigint,COUNT( gender#2) :
bigint]
scala > joinGp. show
15/04/03 17:11:16 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 17:11:16 INFOmapred. FileInputFormat:Total input paths to process:1
name                  MAX( age#0L) COUNT( gender#2)
Personnel Dept        30           2
Testing Department    26           2
Development Dept      33           2
```

这里使用了上例中基于“deptId”列对 people 与 dept 进行 join 的结果，即 joinP。

通过对 joinP 调用 groupBy 方法，在联合结果上继续根据 dept 的“name”列进行分组，并在分组后对指定的列执行指定的聚合操作，这里对“age”列求最大值，对“count”列进行计数。

最后显示的结果上，按部门名称，统计部门员工最大的年龄以及部门的员工的性别（只有“female”，“male”两种）。

16. 保存为表

在对各个 DataFrame 实例进行操作后，获取了目标信息，如果后续需要这些信息的话，就必须执行持久化操作，即将文件保存到存储系统或表中。

下面给出几种持久化的案例。

1）首先将实例持久化到表中。

```
scala > jionP. saveAsTable( " peopleDeplJion" )
```

```
15/04/03 17:05:56 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = peop
ledepljion
15/04/03 17:05:56 INFOHiveMetaStore. audit:ugi = harli
ip = unknown - ip - addrcmd = get_table:db = default tbl = peopledepljion
15/04/03 17:05:56 INFOmetastore. HiveMetaStore:0:get_database:default
15/04/03 17:05:56 INFOHiveMetaStore. audit:ugi = harli
ip = unknown - ip - addrcmd = get_database:default
15/04/03 17:05:56 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = peopledepljion
15/04/03 17:05:56 INFOHiveMetaStore. audit:ugi = harli
ip = unknown - ip - addrcmd = get_table:db = default tbl = peopledepljion
SLF4J:Failed to load class "org. slf4j. impl. StaticLoggerBinder".
SLF4J:Defaulting to no - operation(NOP)logger implementation
SLF4J:See http://www. slf4j. org/codes. html#StaticLoggerBinder for further details.
15/04/03 17:05:57 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 17:05:57 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 17:06:09 INFOhadoop. ParquetFileReader:Initiating action with parallelism:5
15/04/03 17:06:11 INFOmetastore. HiveMetaStore:0:create_table:Table(tableName:peopledepljion, dbName:default, owner:harli, createTime:1428051971, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:[FieldSchema(name:col, type:array<string>, comment:from deserializer)], location:null, inputFormat:org. apache. hadoop. mapred. SequenceFileInputFormat, outputFormat:org. apache. hadoop. hive. ql. io. HiveSequenceFileOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null, serializationLib:org. apache. hadoop. hive. serde2. MetadataTypedColumnsetSerDe, parameters:{serialization. format = 1, path = hdfs://wxx214:9000/user/hive/warehouse/peopledepljion}), bucketCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNames:[], skewedColValues:[], skewedColValueLocationMaps:{})), partitionKeys:[], parameters:{spark. sql. sources. schema. part. 0 = {"type":"struct","fields":[{"name":"age","type":"long","nullable":true,"metadata":{}},{"name":"deptId","type":"long","nullable":true,"metadata":{}},{"name":"gender","type":"string","nullable":true,"metadata":{}},{"name":"job number","type":"string","nullable":true,"metadata":{}},{"name":"name","type":"string","nullable":true,"metadata":{}},{"name":"salary","type":"long","nullable":true,"metadata":{}},{"name":"id","type":"long","nullable":true,"metadata":{}},{"name":"name","type":"string","nullable":true,"metadata":{}}]}, EXTERNAL = FALSE, spark. sql. sources. schema. numParts = 1, spark. sql. sources. provider = org. apache. spark. sql. parquet}, viewOriginalText:null, viewExpandedText:null, tableType:MANAGED_TABLE)
15/04/03 17:06:11 INFOHiveMetaStore. audit:ugi = harli    ip = unknown - ip - addrcmd = create_table:Table(tableName:peopledepljion, dbName:default, owner:harli, createTime:1428051971, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:[FieldSchema(name:col, type:array<string>, comment:from deserializer)], location:null, inputFormat:org. apache. hadoop. mapred. SequenceFileInputFormat, outputFormat:org. apache. hadoop. hive. ql. io. HiveSequenceFileOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null, serializationLib:org. apache. hadoop. hive. serde2. MetadataTypedColumnsetSerDe, parameters:{serialization. format = 1, path = hdfs://wxx214:9000/user/hive/warehouse/peopledepljion}), bucketCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNames:[], skewedColValues:[], skewedColValueLocationMaps:{})), partitionKeys:[], parameters:{spark. sql. sources. schema. part. 0 = {"type":"struct","fields":[{"name":"age","type":"long","nullable":true,"metadata":{}},{"name":"deptId","type":"long","nullable":true,"metadata":{}},{"name":"gender","type":"string","nullable":true,"metadata":{}},{"name":"job number","type":"string","nullable":true,"metadata":{}},{"name":"name","type":"string","nullable":true,"metadata":{}},{"name":"salary","type":"long","nullable":true,"metadata":{}},{"name":"id","type":"long","nullable":true,"metadata":{}},{"
```

```
name":"name","type":"string","nullable":true,"metadata":{}}]}, EXTERNAL = FALSE,
spark.sql.sources.schema.numParts = 1, spark.sql.sources.provider = org.apache.spark.sql.parquet},
viewOriginalText:null, viewExpandedText:null, tableType:MANAGED_TABLE)
15/04/03 17:06:11 INFO hive.log:Updating table stats fast forpeopledepljion
15/04/03 17:06:11 INFO hive.log:Updated size of tablepeopledepljion to 154795
15/04/03 17:06:11 INFOmetastore.HiveMetaStore:0:get_table:db = default tbl = peopledepljion
15/04/03 17:06:11 INFOHiveMetaStore.audit:ugi = harli    ip = unknown - ip - addrcmd = get_table:db
 = default tbl = peopledepljion
```

此时，使用的默认的 hive，没有连接到现有的 hive 环境上。

通过调用 DataFrame 的 saveAsTable 方法，将实例持久化到 hive 的“peopleDeplJion”表中。对应地，会在 HDFS 上构建 hive 用户的目录，即/user/hive，同时生成 hive 的仓库目录，即/user/hive/warehouse，每个构建的 hive 表，都会对应到该仓库下的一个子目录，持久化 DataFrame 实例后，对应创建列“peopleDeplJion”这个子目录。

通过 Web Interface 界面(http://namenode:50070)查看目录结构，如图 3.3 所示。其中 namenode 为启动 NameNode 进程的节点，当前环境下的节点地址为 192.168.70.214。

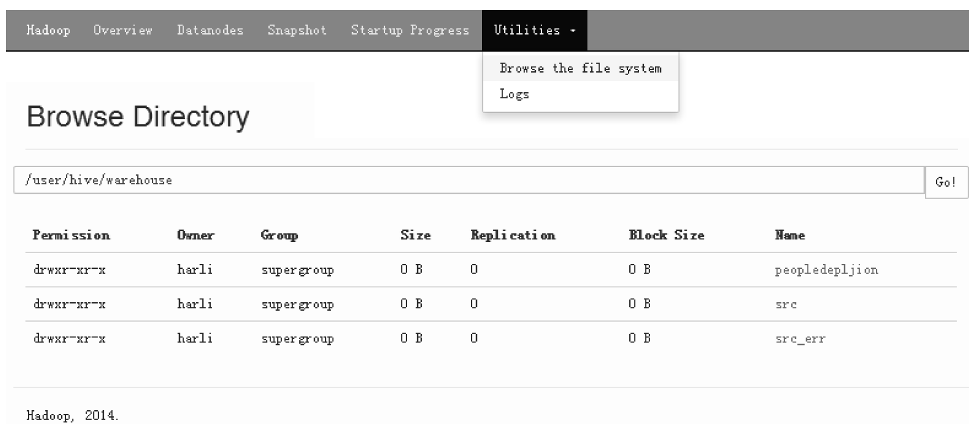


图 3.3　Hadoop 文件系统持久化后的界面

表相关的操作还有 registerTempTable 方法，这部分将在下一章节进行分析。

2）保存为 json 文件

```
scala > hsqlDF.save("/user/harli/hsqlDF.json","json")
15/04/03 17:20:34 INFO Configuration.deprecation:mapred.tip.id is deprecated. Instead, use mapre-
duce.task.id
15/04/03 17:20:34 INFO Configuration.deprecation:mapred.task.id is deprecated. Instead, use mapre-
duce.task.attempt.id
15/04/03 17:20:34 INFO Configuration.deprecation:mapred.task.is.map is deprecated. Instead, use
mapreduce.task.ismap
15/04/03 17:20:34 INFO Configuration.deprecation:mapred.task.partition is deprecated. Instead, use
mapreduce.task.partition
15/04/03 17:20:34 INFO Configuration.deprecation:mapred.job.id is deprecated. Instead, use mapre-
duce.job.id
```

这里使用 save 方法，通过在方法中指定数据源为“json”，可以将 DataFrame 实例持久

化到指定的路径上。

通过 Web Interface 界面查看保存为 json 文件的结果，如图 3.4 所示，在/user/harli 路径下生成列指定的 hsqlDF. json 文件。

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Browse Directory

/user/harli Go!

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	harli	supergroup	0 B	0	0 B	hsqlDF.json
drwxr-xr-x	harli	supergroup	0 B	0	0 B	json
drwxr-xr-x	harli	supergroup	0 B	0	0 B	parquet

图 3.4　Hadoop 文件系统保存为 json 后的界面

3）保存为 parquet 文件

```
scala > hsqlDF. save( "/user/harli/hsqlDF. parquet" ," parquet" )
15/04/03 17:21:42 INFOhadoop. ParquetFileReader:Initiating action with parallelism:5
```

这里使用 save 方法，通过在方法中指定数据源为“parquet”，可以将 DataFrame 实例持久化到指定的路径上。

通过 Web Interface 界面查看保存为 parquet 文件的结果，如图 3.5 所示，在/user/harli 路径下生成列指定的 hsqlDF. parquet 文件。

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Browse Directory

/user/harli Go!

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	harli	supergroup	0 B	0	0 B	hsqlDF.json
drwxr-xr-x	harli	supergroup	0 B	0	0 B	hsqlDF.parquet
drwxr-xr-x	harli	supergroup	0 B	0	0 B	json
drwxr-xr-x	harli	supergroup	0 B	0	0 B	parquet

图 3.5　Hadoop 文件系统保存为 parquet 后的界面

3.2.3　DataFrame 与 RDD 之间的转换案例与解析

这部分内容是在上一节的基础上，对某些案例进行扩展并深入分析。其中，部分案例源自官网，在此对其进行深入解析。

一、DataFrame 与 RDD 间的交互

DataFrame 可以从结构化文件、hive 表、外部数据库以及现有的 RDD 加载构建得到。具

体的结构化文件、hive 表、外部数据库的相关加载示例可以参考章节 3.3 Spark SQL 处理各种数据源的案例与解析部分。这里主要针对从现有的 RDD 来构建 DataFrame 进行实践与解析。

Spark SQL 支持两种方法将存在的 RDD 转换为 DataFrame。



1）第一种方法是使用反射来推断包含特定对象类型的 RDD 的模式。在写 Spark 程序的同时，已经知道了模式，这种基于反射的方法可以使代码更简洁并且程序工作得更好。

2）第二种方法是通过一个编程接口来实现，这个接口允许构造一个模式，然后在存在的 RDD 上使用它。虽然这种方法代码较为冗长，但是它允许在运行期之前不知道列以及列的类型的情况下构造 DataFrame。

（一）通过反射机制构建 DataFrame

利用反射推断模式，Spark SQL 的 Scala 接口支持将包含样本类的 RDD 自动转换为 DataFrame。这个样本类定义了表的模式。样本类的参数名字通过反射来读取，然后作为列的名字。样本类可以嵌套或者包含复杂的类型如序列或者数组。这个 RDD 可以隐式转化为一个 DataFrame，然后注册为一个表，表可以在后续的 SQL 语句中使用。

以 people. txt 作为测试数据，使用 Scala 语言来创建 DataFrame。

1. 首先，查看 people. txt 数据

```
[harli@wxx215spark]$hdfs dfs -cat /user/harli/data/people.txt
Michael,33
Andy,30
Justin,16
John,20
Herry,19
```

使用 Hadoop 的 hdfs dfs -cat 命令查看 people. txt 文件内容。

注意： 这里已经将 HADOOP_HOME/bin 添加到环境变量 PATH 中，因此直接使用 hdfs 命令。

2. 定义 people. xml 对应的样本类 People，并加载文件，构建 DataFrame

```
scala>case class People(name:String,age:Int)
defined class People
scala>val rdd=sc.textFile("/user/harli/data/people.txt").map(_.split(",")).map(p=>People(p
(0),p(1).trim.toInt))
rdd:org.apache.spark.rdd.RDD[People]=MapPartitionsRDD[35] at map at <console>:24
scala>rdd.collect
15/04/06 08:12:26 INFOmapred.FileInputFormat:Total input paths to process:1
res9:Array[People] = Array(People(Michael,33),People(Andy,30),People(Justin,16),People
(John,20),People(Herry,19))
scala>val people=rdd.toDF()
people:org.apache.spark.sql.DataFrame=[name:string,age:int]
scala>people.show
name     age
Michael  33
Andy     30
Justin   16
John     20
Herry    19
```

示例中：

1）首先定义样本类 People。

2）之后通过 SparkContext 的 textFile 方法将文件加载进来，用“,”作为分隔符，将每一行数据分割为包含两个元素（“name” 和 “age”）的数组，继续使用 map 方法将数组映射成样本类 People，此时可以得到对应该文件的 RDD 实例，其元素类型为样本类 People。

3）最后在 RDD 实例上使用 toDF 方法，转换为对应的 DataFrame 实例 people。

注意：在以 spark - submit 方式提交的应用程序中，不要将样本类 People 定义在和 SparkContext 实例相同的作用域中，即需要将样本类定义在 main 方法外或 object 外。

用 Scala 函数范式修改以上代码：

```
scala > val people = sc. textFile( "/user/harli/data/people. txt" ). map( _. split( "," ) ). map( p = > People( p(0) ,p(1). trim. toInt) ). toDF( )
people:org. apache. spark. sql. DataFrame = [ name:string,age:int]
```

4）将 DataFrame 注册成临时表，并对表执行 SQL 查询语句。

```
scala > people. registerTempTable( "people" )
//查看当前数据库信息
scala > sqlContext. sql( " show databases" ). show
result
default
//查看当前的全部表名
scala > sqlContext. tableNames
res4:Array[ String] = Array( people,pokes,pokes_test,t)
//获取"people",并返回 DataFrame
scala > sqlContext. table( "people" )
res5:org. apache. spark. sql. DataFrame = [ age:string,name:string]
//查看当前"default"数据库中的表,返回类型为 DataFrame
scala > sqlContext. tables( "default" )
res6:org. apache. spark. sql. DataFrame = [ tableName:string,isTemporary:boolean]
scala > sqlContext. tables( "default" ). show
tableNameisTemporary
people      true
pokes       false
pokes_test false
t           false
//SQLContext 的 sql 函数使应用程序可以用编程方式运行 sql 查询
//并返回结果 DataFrame 实例
scala > val teenagers = sqlContext. sql( "SELECT name FROM people WHERE age > = 13 AND age < = 19" )
15/04/06 08:42:01 WARN conf. HiveConf:DEPRECATED:Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/04/06 08:42:01 INFO parse. ParseDriver:Parsing command:SELECT name FROM people WHERE age > = 13 AND age < = 19
15/04/06 08:42:01 INFO parse. ParseDriver:Parse Completed
teenagers:org. apache. spark. sql. DataFrame = [ name:string]
```

```
//用 DataFrame 的 map 方法,转换每一条记录 t,t 的列可以通过下标来访问
scala > teenagers. map( t => "Name:" + t(0) ). collect( ). foreach( println)
15/04/06 08:42:13 INFOmapred. FileInputFormat:Total input paths to process:1
Name:Justin
Name:Herry
```



调用 people 的 registerTempTable 方法，注册为临时表“people”，注册后就可以通过 SQL 方法使用 sqlContext 支持的 SQL 语句对该临时表进行操作。示例中使用“SELECT”语句从表“people”中查找出年龄在 13 到 19 之间的人员的名字。

SQL 查询语句得到的结果类型是 DataFrame 实例，支持所有普通 RDD 的全部操作。示例中最后一行代码将 DataFrame 的每一条记录用 map 方法转换为字符串，字符串中提取了“name”列，然后用 collect 方法收集记录并在界面打印出来。可以单步执行这一行代码，查看每个 API 操作的返回类型。

（二）用编程指定模式构建 DataFrame

当样本类不能提前确定（例如，记录的结构是经过编码的字符串，或者一个文本集合将会被解析，不同的字段投影给不同的用户），一个 DataFrame 里实例可以通过下面三个步骤来创建。

1）从原来的 RDD 创建一个元素类型为行（Row）的 RDD。

2）创建由一个 StructType 表示的模式（Schema），与第一步创建的 RDD 的 Row 的结构相匹配。

3）在元素类型为 Row 的 RDD 上，通过 applySchema 方法应用第二步构建的 Schema。

Scala 语言创建 DataFrame 的方式如下：

1. 加载文件，并构建文件对应的 Schema 的列名字符串

```
//构建一个 RDD 实例 people
scala > val people = sc. textFile( "/user/harli/data/people. txt" )
people:org. apache. spark. rdd. RDD [ String ] =/user/harli/data/people. txt MapPartitionsRDD [ 47 ] at
textFile at < console > :22
//指定一个 Schema 包含的列名
scala > val schemaString = "name age"
schemaString:String = name age
```

2. 导入 org. apache. spark. sql. types. _，构建 Schema 信息

```
scala > import org. apache. spark. sql. types. _
import org. apache. spark. sql. types. _
//构建一个 Schema,为每一列构建一个 StructField,这里的 split 分隔符
//和前面定义列名时一致,都是空格。
//指定每一列都为 StringType 类型,且可以为 null
scala > val schema =
     |  StructType(
     |    schemaString. split(" "). map( fieldName => StructField( fieldName,StringType,true) ) )
schema: org. apache. spark. sql. types. StructType = StructType ( StructField ( name, StringType, true ),
StructField( age,StringType,true) )
```

注意：官网示例中的 import 语句有问题，StructType 等类型在 org. apache. spark. sql. types 中。

3. 将 RDD 的字符串元素转换为 DataFrame 的 Row 类型

```
//将 RDD 中的每条记录转换成一个行(Row)实例
scala > val rowRDD = people. map(_. split(",")). map(p => Row(p(0),p(1). trim))
rowRDD:org. apache. spark. rdd. RDD[org. apache. spark. sql. Row] = MapPartitionsRDD[49] at map at <console>:35
```

4. 应用之前定义的 Schema，创建 DataFrame

```
scala > val peopleDataFrame = sqlContext. createDataFrame(rowRDD,schema)
peopleDataFrame:org. apache. spark. sql. DataFrame = [name:string,age:string]
```

5. 注册到临时表中，并通过 SQL 查询语句从该临时表中构建出新的 DataFrame 实例

```
scala > people. registerTempTable("people")
scala > val results = sqlContext. sql("SELECT name FROM people")
15/04/06 08:59:24 WARN conf. HiveConf:DEPRECATED:Configuration property hive. metastore. local no longer has any effect.  Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/04/06 08:59:24 INFO parse. ParseDriver:Parsing command:SELECT name FROM people
15/04/06 08:59:24 INFO parse. ParseDriver:Parse Completed
results:org. apache. spark. sql. DataFrame = [name:string]
```

6. 将新构建的 results 的内容输出到控制台

```
scala > results. map(t => "Name:" + t(0)). collect(). foreach(println)
15/04/06 08:59:38 INFOmapred. FileInputFormat:Total input paths to process:1
Name:Michael
Name:Andy
Name:Justin
Name:John
Name:Herry
```

（三）DataFrame 与 RDD 间的关系

构建一个 DataFrame，然后调用 rdd 方法，用 toDebugString 查看 rdd 的 Lineage 关系。

```
//定义样本类,使用 spark - submit 提交应用时,定义放到 main 方法或 object 外面
//不要和样本类的使用代码放在相同的作用域内
scala > case class People(name:String,age:Int)
defined class People
//加载文件,并将每一行 split 后构建 People 实例(对应调用了 People 的 apply 方法)
scala > val people = sc. textFile("/user/harli/data/people. txt"). map(_. split(",")). map(p => People(p(0),p(1). trim. toInt)). toDF()
people:org. apache. Spark. sql. DataFrame = [name:string,age:int]
//查看 rdd 的 Lineage 关系
scala > people. rdd. toDebugString
15/04/06 09:12:05 INFOmapred. FileInputFormat:Total input paths to process:1
res2:String =
(2)MapPartitionsRDD[10] at map at DataFrame. scala:889 []
  |  MapPartitionsRDD[9] at mapPartitions at ExistingRDD. scala:35 []
  |  MapPartitionsRDD[8] at map at <console>:24 []
  |  MapPartitionsRDD[7] at map at <console>:24 []
  |  /user/harli/data/people. txt MapPartitionsRDD[6] at textFile at <console>:24 []
```

```
| /user/harli/data/people.txt HadoopRDD[5] at textFile at <console>:24 []
```

从 MapPartitionsRDD[10] at map at DataFrame.scala:889 []这一行，可以看到从 DataFrame 转换得到 RDD 时（即这里的 people.rdd 调用），实际上内部是调用了 DataFrame 的 map 方法，将 DataFrame 的 Row 转换为 RDD 的元素。这可以参考前面构建 DataFrame 的三个步骤。

3.2.4 缓存表（列式存储）的案例与解析

这部分内容可用于性能调优，通过将数据缓存到内存来提高性能。

Spark SQL 可以通过调用 sqlContext.cacheTable("tableName")方法来缓存使用柱状格式的表。然后，Spark 将会仅仅浏览需要的列并且自动地压缩数据以减少内存的使用以及垃圾回收的压力。可以通过调用 sqlContext.uncacheTable("tableName")方法在内存中删除表。

注意： 如果调用 schemaRDD.cache()而不是 sqlContext.cacheTable（...），表将不会用柱状格式来缓存（即列式存储）。在这种情况下，强烈推荐调用 sqlContext.cacheTable（...）。

可以在 SQLContext 上使用 setConf 方法或者在用 SQL 时运行 "SET key = value" 命令来配置内存缓存，属性部分配置信息如表 3.1 所示。

表 3.1 用"SET key = value"命令的部分属性配置

属性名称	默认值	含义
spark.sql.inMemoryColumnarStorage.compressed	true	当设置为 true 时，Spark SQL 将为基于数据统计信息的每列自动选择一个压缩算法
spark.sql.inMemoryColumnarStorage.batchSiz	10000	柱状缓存的批数据大小。更大的批数据可以提高内存的利用率以及压缩效率，但有 OOM 的风险

对表进行缓存的示例：

1. 用 sql 方法从 people 中查找数据

```
scala>val results = sqlContext.sql("SELECT name FROM people")
15/04/06 09:27:04 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/04/06 09:27:04 INFO parse.ParseDriver:Parsing command:SELECT name FROM people
15/04/06 09:27:04 INFO parse.ParseDriver:Parse Completed
results:org.apache.spark.sql.DataFrame = [name:string]
```

这里的 people 是在前面构建的 DataFrame 中注册的表。

2. 对表进行缓存

```
scala>sqlContext.cacheTable("people")
scala>results.show
15/04/06 09:24:09 INFO spark.SparkContext:Starting job:runJob at SparkPlan.scala:121
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Got job 12(runJob at SparkPlan.scala:121)with 1 output partitions(allowLocal = false)
```

```
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Final stage:Stage 12(runJob at SparkPlan.scala:121)
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Parents of final stage:List()
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Missing parents:List()
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Submitting Stage 12(MapPartitionsRDD[26] at map at SparkPlan.scala:96),which has no missing parents
15/04/06 09:24:09 INFO storage.MemoryStore:ensureFreeSpace(9480) called with curMem=313045, maxMem=280248975
15/04/06 09:24:09 INFO storage.MemoryStore:Block broadcast_14 stored as values in memory(estimated size 9.3 KB,free 267.0 MB)
15/04/06 09:24:09 INFO storage.MemoryStore:ensureFreeSpace(6065) called with curMem=322525, maxMem=280248975
15/04/06 09:24:09 INFO storage.MemoryStore:Block broadcast_14_piece0 stored as bytes in memory (estimated size 5.9 KB,free 267.0 MB)
15/04/06 09:24:09 INFO storage.BlockManagerInfo:Added broadcast_14_piece0 in memory on cluster01:41777(size:5.9 KB,free:267.2 MB)
15/04/06 09:24:09 INFO storage.BlockManagerMaster:Updated info of block broadcast_14_piece0
15/04/06 09:24:09 INFO spark.SparkContext:Created broadcast 14 from broadcast at DAGScheduler.scala:839
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Submitting 1 missing tasks from Stage 12(MapPartitionsRDD[26] at map at SparkPlan.scala:96)
15/04/06 09:24:09 INFO scheduler.TaskSchedulerImpl:Adding task set 12.0 with 1 tasks
15/04/06 09:24:09 INFO scheduler.TaskSetManager:Starting task 0.0 in stage 12.0(TID 12,cluster01,NODE_LOCAL,1312 bytes)
15/04/06 09:24:09 INFO storage.BlockManagerInfo:Added broadcast_14_piece0 in memory on cluster01:34364(size:5.9 KB,free:267.2 MB)
15/04/06 09:24:09 INFO storage.BlockManagerInfo:Added rdd_24_0 in memory on cluster01:34364 (size:496.0 B,free:267.2 MB)
15/04/06 09:24:09 INFO scheduler.TaskSetManager:Finished task 0.0 in stage 12.0(TID 12)in 228 ms on cluster01(1/1)
15/04/06 09:24:09 INFO scheduler.TaskSchedulerImpl:Removed TaskSet 12.0,whose tasks have all completed,from pool
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Stage 12(runJob at SparkPlan.scala:121)finished in 0.229 s
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Job 12 finished:runJob at SparkPlan.scala:121, took 0.240746 s
15/04/06 09:24:09 INFO spark.SparkContext:Starting job:runJob at SparkPlan.scala:121
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Got job 13(runJob at SparkPlan.scala:121)with 1 output partitions(allowLocal=false)
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Final stage:Stage 13(runJob at SparkPlan.scala:121)
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Parents of final stage:List()
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Missing parents:List()
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Submitting Stage 13(MapPartitionsRDD[26] at map at SparkPlan.scala:96),which has no missing parents
15/04/06 09:24:09 INFO storage.MemoryStore:ensureFreeSpace(9480) called with curMem=328590, maxMem=280248975
15/04/06 09:24:09 INFO storage.MemoryStore:Block broadcast_15 stored as values in memory(estimated size 9.3 KB,free 266.9 MB)
```

```
15/04/06 09:24:09 INFO storage.MemoryStore:ensureFreeSpace(6065) called with curMem=338070,
maxMem=280248975
15/04/06 09:24:09 INFO storage.MemoryStore:Block broadcast_15_piece0 stored as bytes in memory
(estimated size 5.9 KB,free 266.9 MB)
15/04/06 09:24:09 INFO storage.BlockManagerInfo:Added broadcast_15_piece0 in memory on cluster01:41777 (size:5.9 KB,free:267.2 MB)
15/04/06 09:24:09 INFO storage.BlockManagerMaster:Updated info of block broadcast_15_piece0
15/04/06 09:24:09 INFO spark.SparkContext:Created broadcast 15 from broadcast at DAGScheduler.scala:839
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Submitting 1 missing tasks from Stage 13 (MapPartitionsRDD[26] at map at SparkPlan.scala:96)
15/04/06 09:24:09 INFO scheduler.TaskSchedulerImpl:Adding task set 13.0 with 1 tasks
15/04/06 09:24:09 INFO scheduler.TaskSetManager:Starting task 0.0 in stage 13.0 (TID 13,cluster01,NODE_LOCAL,1312 bytes)
15/04/06 09:24:09 INFO storage.BlockManagerInfo:Added broadcast_15_piece0 in memory on cluster01:34364 (size:5.9 KB,free:267.2 MB)
15/04/06 09:24:09 INFO storage.BlockManagerInfo:Added rdd_24_1 in memory on cluster01:34364
(size:480.0 B,free:267.2 MB)
15/04/06 09:24:09 INFO scheduler.TaskSetManager:Finished task 0.0 in stage 13.0 (TID 13) in 71
ms on cluster01 (1/1)
15/04/06 09:24:09 INFO scheduler.TaskSchedulerImpl:Removed TaskSet 13.0,whose tasks have all
completed,from pool
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Stage 13 (runJob at SparkPlan.scala:121) finished
in 0.072 s
15/04/06 09:24:09 INFO scheduler.DAGScheduler:Job 13 finished:runJob at SparkPlan.scala:121,
took 0.102063 s
name
Michael
Andy
Justin
John
Herry
```

cacheTable 操作是惰性的，在调用 sqlContext.cacheTable("people")后，并不会马上进行缓存，执行完此操作后对应的 Web Interface 界面（http://driver:4040）如图 3.6 所示。

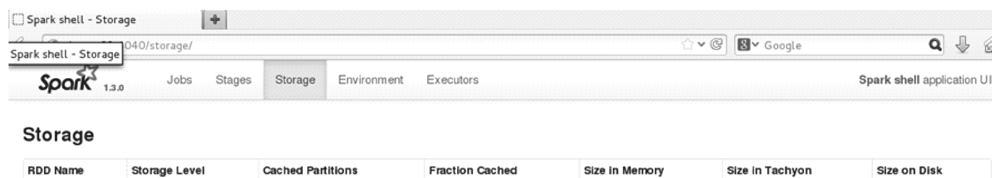


图 3.6　Driver 界面中缓存表前的 Storage 界面

通过 results 的 show 方法，触发缓存操作，这时仅仅将 results 对应的“name”列缓存内存中。对应的 Web Interface 界面（http://driver:4040）如图 3.7 所示。

当前的两个分区都已经全部缓存到列内存中。

缓存成功后，继续调用 show 方法，显示 results。对于结果中时间上的变化，这里是由于数据量比较小，缓存后的时间消耗只减少了一点。代码如下。

cluster01:4040/storage/

Spark 1.3.0 Jobs Stages Storage Environment Executors Spark shell application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
In-memory table people	Memory Deserialized 1x Replicated	2	100%	976.0 B	0.0 B	0.0 B

图 3.7　Driver 界面中缓存表触发后的 Storage 界面

scala > results. show

15/04/06 09:24:38 INFO spark. SparkContext:Starting job:runJob at SparkPlan. scala:121

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Got job 18 (runJob at SparkPlan. scala:121) with 1 output partitions (allowLocal = false)

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Final stage:Stage 18(runJob at SparkPlan. scala:121)

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Parents of final stage:List()

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Missing parents:List()

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Submitting Stage 18 (MapPartitionsRDD[32] at map at SparkPlan. scala:96),which has no missing parents

15/04/06 09:24:38 INFO storage. MemoryStore:ensureFreeSpace(9872) called with curMem = 378797, maxMem = 280248975

15/04/06 09:24:38 INFO storage. MemoryStore:Block broadcast_20 stored as values in memory (estimated size:9.6 KB,free:266.9 MB)

15/04/06 09:24:38 INFO storage. MemoryStore:ensureFreeSpace(6564) called with curMem = 388669, maxMem = 280248975

15/04/06 09:24:38 INFO storage. MemoryStore:Block broadcast_20_piece0 stored as bytes in memory (estimated size 6.4 KB,free:266.9 MB)

15/04/06 09:24:38 INFO storage. BlockManagerInfo:Added broadcast_20_piece0 in memory on cluster01:41777 (size:6.4 KB,free:267.2 MB)

15/04/06 09:24:38 INFO storage. BlockManagerMaster:Updated info of block broadcast_20_piece0

15/04/06 09:24:38 INFO spark. SparkContext:Created broadcast 20 from broadcast at DAGScheduler. scala:839

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Submitting 1 missing tasks from Stage 18 (MapPartitionsRDD[32] at map at SparkPlan. scala:96)

15/04/06 09:24:38 INFO scheduler. TaskSchedulerImpl:Adding task set 18.0 with 1 tasks

15/04/06 09:24:38 INFO scheduler. TaskSetManager:Starting task 0.0 in stage 18.0 (TID 18, cluster01,PROCESS_LOCAL,1312 bytes)

15/04/06 09:24:38 INFO storage. BlockManagerInfo:Added broadcast_20_piece0 in memory on cluster01:34364 (size:6.4 KB,free:267.2 MB)

15/04/06 09:24:38 INFO scheduler. TaskSetManager:Finished task 0.0 in stage 18.0 (TID 18) in 45 ms on cluster01 (1/1)

15/04/06 09:24:38 INFO scheduler. TaskSchedulerImpl:Removed TaskSet 18.0,whose tasks have all completed,from pool

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Stage 18 (runJob at SparkPlan. scala:121) finished in 0.046 s

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Job 18 finished:runJob at SparkPlan. scala:121, took 0.057244 s

15/04/06 09:24:38 INFO spark. SparkContext:Starting job:runJob at SparkPlan. scala:121

15/04/06 09:24:38 INFO scheduler. DAGScheduler:Got job 19 (runJob at SparkPlan. scala:121) with 1 output partitions (allowLocal = false)



```
15/04/06 09:24:38 INFO scheduler.DAGScheduler:Final stage:Stage 19(runJob at SparkPlan.scala:121)
15/04/06 09:24:38 INFO scheduler.DAGScheduler:Parents of final stage:List()
15/04/06 09:24:38 INFO scheduler.DAGScheduler:Missing parents:List()
15/04/06 09:24:38 INFO scheduler.DAGScheduler:Submitting Stage 19 (MapPartitionsRDD[32] at map at SparkPlan.scala:96),which has no missing parents
15/04/06 09:24:38 INFO storage.MemoryStore:ensureFreeSpace(9872) called with curMem=395233,maxMem=280248975
15/04/06 09:24:38 INFO storage.MemoryStore:Block broadcast_21 stored as values in memory (estimated size 9.6 KB,free 266.9 MB)
15/04/06 09:24:38 INFO storage.MemoryStore:ensureFreeSpace(6564) called with curMem=405105,maxMem=280248975
15/04/06 09:24:38 INFO storage.MemoryStore:Block broadcast_21_piece0 stored as bytes in memory (estimated size 6.4 KB,free 266.9 MB)
15/04/06 09:24:38 INFO storage.BlockManagerInfo:Added broadcast_21_piece0 in memory on cluster01:41777 (size:6.4 KB,free:267.2 MB)
15/04/06 09:24:38 INFO storage.BlockManagerMaster:Updated info of block broadcast_21_piece0
15/04/06 09:24:38 INFO spark.SparkContext:Created broadcast 21 from broadcast at DAGScheduler.scala:839
15/04/06 09:24:38 INFO scheduler.DAGScheduler:Submitting 1 missing tasks from Stage 19 (MapPartitionsRDD[32] at map at SparkPlan.scala:96)
15/04/06 09:24:38 INFO scheduler.TaskSchedulerImpl:Adding task set 19.0 with 1 tasks
15/04/06 09:24:38 INFO scheduler.TaskSetManager:Starting task 0.0 in stage 19.0 (TID 19,cluster01,PROCESS_LOCAL,1312 bytes)
15/04/06 09:24:38 INFO storage.BlockManagerInfo:Added broadcast_21_piece0 in memory on cluster01:34364 (size:6.4 KB,free:267.2 MB)
15/04/06 09:24:38 INFO scheduler.TaskSetManager:Finished task 0.0 in stage 19.0 (TID 19) in 51 ms on cluster01 (1/1)
15/04/06 09:24:38 INFO scheduler.TaskSchedulerImpl:Removed TaskSet 19.0,whose tasks have all completed,from pool
15/04/06 09:24:38 INFO scheduler.DAGScheduler:Stage 19 (runJob at SparkPlan.scala:121) finished in 0.052 s
15/04/06 09:24:38 INFO scheduler.DAGScheduler:Job 19 finished:runJob at SparkPlan.scala:121,took 0.059765 s
name
Michael
Andy
Justin
John
Herry
```

调用 uncacheTable 方法释放缓存：

```
scala>sqlContext.uncacheTable("people")
15/04/06 09:36:22 INFOrdd.MapPartitionsRDD:Removing RDD 24 from persistence list
15/04/06 09:36:22 INFO storage.BlockManager:Removing RDD 24
```

调用后，可以查看 Web Interface 界面，内存会马上释放。

3.2.5 DataFrame API 的应用案例与分析

针对相同功能的 API 进行分组，基本上以官方网站上 API 的顺序给出应用案例。

一、collect 与 collectAsList

1. 定义

```
def collect( ):Array[ Row]
def collectAsList( ):List[ Row]
```

2. 功能描述

collect 返回一个数组，包含 DataFrame 中包含的全部 Rows。

collectAsList 返回一个 Java List，包含 DataFrame 中包含的全部 Rows。

3. 示例

```
scala > val df = sqlContext. jsonFile( "/user/harli/test/people. json" )
15/04/04 08:06:23 INFOmapred. FileInputFormat:Total input paths to process: 1
df:org. apache. spark. sql. DataFrame = [ age: bigint, deptId: bigint, gender: string, job  number: string,
name:string,salary:bigint]
scala > df. show
15/04/04 08:07:17 INFOmapred. FileInputFormat:Total input paths to process: 1
agedeptId gender job number name       salary
33   1        male     001          Michael    3000
30   2        female   002          Andy       4000
19   3        male     003          Justin     5000
32   1        male     004          John       6000
20   2        female   005          Herry      7000
26   3        male     006          Jack       3000
scala > df. collect
15/04/04 08:06:30 INFOmapred. FileInputFormat:Total input paths to process:1
res3:Array[ org. apache. spark. sql. Row] = Array( [33,1,male,001,Michael,3000],[30,2,female,
002,Andy,4000],[19,3,male,003,Justin,5000],[32,1,male,004,John,6000],[20,2,female,005,
Herry,7000],[26,3,male,006,Jack,3000])
scala > df. collectAsList
res5:java. util. List[ org. apache. spark. sql. Row] = [[33,1,male,001,Michael,3000],[30,2,female,
002,Andy,4000],[19,3,male,003,Justin,5000],[32,1,male,004,John,6000],[20,2,female,005,
Herry,7000],[26,3,male,006,Jack,3000]]
```

4. 示例解析

案例中首先加载列 people. json 文件，然后以表格形式呈现其内容。

两个 collect 型的方法都可以获取 df 的全部 Rows 数据，只是返回的类型不同。调用 collect 方法，返回的是数组，数组元素的类型为 org. apache. spark. sql. Row；调用 collectAslist，返回类型 java. util. List[org. apache. spark. sql. Row]。

二、count

1. 定义

```
def count( ):Long
```

2. 功能描述

返回 DataFrame 的 rows 个数。

3. 示例



```
scala > df. count
15/04/04 08:14:16 INFOmapred. FileInputFormat:Total input paths to process:1
res6:Long = 6
```

三、first

1. 定义

```
def first( ):Row
```

2. 功能描述

返回 DataFrame 的第一个 row。

3. 示例

```
scala > df. first
15/04/04 08:16:04 INFOmapred. FileInputFormat:Total input paths to process:1
res7:org. apache. Spark. sql. Row = [33,1,male,001,Michael,3000]
```

四、head

1. 定义

```
def head( ):Row
def head(n:Int):Array[Row]
```

2. 功能描述

不带参数的 head 方法，返回 DataFrame 的第一个 Row。指定参数 n 时，则返回前 n 个 Rows。

3. 示例

```
scala > df. head
15/04/04 08:16:10 INFOmapred. FileInputFormat:Total input paths to process:1
res8:org. apache. spark. sql. Row = [33,1,male,001,Michael,3000]
scala > df. head(4)
15/04/04 08:16:15 INFOmapred. FileInputFormat:Total input paths to process:1
res9:Array[org. apache. spark. sql. Row] = Array([33,1,male,001,Michael,3000],[30,2,female,
002,Andy,4000],[19,3,male,003,Justin,5000],[32,1,male,004,John,6000])
```

五、show

1. 定义

```
def show( ):Unit
def show(numRows:Int):Unit
```

2. 功能描述

不带参数的 show 方法，以表格形式显示 DataFrame 的前 20 个 Rows。指定参数 numRows 时，则显示指定 numRows 个数的 Rows。

3. 示例

```
scala > df. show
15/04/04 08:17:04 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary
33   1        male     001          Michael   3000
30   2        female   002          Andy      4000
19   3        male     003          Justin    5000
32   1        male     004          John      6000
20   2        female   005          Herry     7000
26   3        male     006          Jack      3000
scala > df. show(4)
15/04/04 08:17:08 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name       salary
33   1        male     001          Michael   3000
30   2        female   002          Andy      4000
19   3        male     003          Justin    5000
32   1        male     004          John      6000
```

六、take

1. 定义

```
def take(n:Int):Array[Row]
```

2. 功能描述

返回 DataFrame 中指定的前 n 个 Rows。

3. 示例

```
scala > df. take(4)
15/04/04 08:21:41 INFOmapred. FileInputFormat:Total input paths to process:1
res14:Array[org. apache. spark. sql. Row] = Array([33,1,male,001,Michael,3000],[30,2,female,
002,Andy,4000],[19,3,male,003,Justin,5000],[32,1,male,004,John,6000])
```

七、cache

1. 定义

```
def cache():DataFrame. this. type
```

2. 功能描述

将 DataFrame 缓存到内存中。

3. 示例

```
scala > df. cache
res15:df. type = [age:bigint,deptId:bigint,gender:string,job number:string,name:string,salary:bigint]
scala > df. count
res16:Long = 6
scala > df. take(6)
res17:Array[org. apache. spark. sql. Row] = Array([33,1,male,001,Michael,3000],[30,2,female,
```

```
002,Andy,4000],[19,3,male,003,Justin,5000],[32,1,male,004,John,6000],[20,2,female,005,
Herry,7000],[26,3,male,006,Jack,3000])
```

可以看到，当执行了 cache 操作后，对 DataFrame 执行操作时，不需要再从原始数据源加载数据。



八、Columns

1. 定义

```
def columns:Array[String]
```

2. API 的功能描述

以数组形式返回 DataFrame 的全部列名。

3. 示例

```
scala > df. columns
res18:Array[String] = Array(age,deptId,gender,job number,name,salary)
```

九、dtypes

1. 定义

```
def dtypes:Array[(String,String)]
```

2. 功能描述

以数组形式返回 DataFrame 的所有列名及其对应数据类型。

3. 示例

```
scala > df. dtypes
res20:Array[(String,String)] = Array((age,LongType),(deptId,LongType),(gender,StringType),
(job number,StringType),(name,StringType),(salary,LongType))
```

十、explain

1. 定义

```
def explain():Unit
def explain(extended:Boolean):Unit
```

2. 功能描述

这两个方法用于调试目的，不带参数时，仅将 DataFrame 的物理计划打印到控制台上；当指定参数 extended 为 true 时，打印所有计划到控制台上，包括物理计划、逻辑计划。

3. 示例

```
scala > df. explain
 == Physical Plan ==
InMemoryColumnarTableScan [age#6L,deptId#7L,gender#8,job number#9,name#10,salary#11L],[],
(InMemoryRelation [age#6L,deptId#7L,gender#8,job number#9,name#10,salary#11L],true,10000,
StorageLevel(true,true,false,true,1),(PhysicalRDD [age#6L,deptId#7L,gender#8,job number#9,
name#10,salary#11L],MapPartitionsRDD[19] at map at JsonRDD. scala:41),None)
scala > df. explain(true)
 == Parsed Logical Plan ==
Relation[age#6L,deptId#7L,gender#8,job number#9,name#10,salary#11L] JSONRelation(/user/harli/
```

```
data/people.json,1.0,None)
 == Analyzed Logical Plan ==
Relation[age#6L,deptId#7L,gender#8,job number#9,name#10,salary#11L] JSONRelation(/user/harli/
data/people.json,1.0,None)
 == Optimized Logical Plan ==
InMemoryRelation [age#6L,deptId#7L,gender#8,job number#9,name#10,salary#11L],true,10000,
StorageLevel(true,true,false,true,1),(PhysicalRDD [age#6L,deptId#7L,gender#8,job number#9,
name#10,salary#11L],MapPartitionsRDD[19] at map at JsonRDD.scala:41),None
 == Physical Plan ==
InMemoryColumnarTableScan [age#6L,deptId#7L,gender#8,job number#9,name#10,salary#11L],[],
(InMemoryRelation [age#6L,deptId#7L,gender#8,job number#9,name#10,salary#11L],true,10000,
StorageLevel(true,true,false,true,1),(PhysicalRDD [age#6L,deptId#7L,gender#8,job number#9,
name#10,salary#11L],MapPartitionsRDD[19] at map at JsonRDD.scala:41),None)
Code Generation:false
 == RDD ==
```

由“Code Generation：false”可以看到，当前代码生成（CG）优化没有开启。

十一、isLocal

1. 定义

```
def isLocal:Boolean
```

2. 功能描述

如果 collect 和 take 方法可以在本地运行（即不需要任何 Spark Executors）时，返回 true。

3. 示例

```
scala> val rdd = sc.parallelize("""{"name":"Yin","address":"beijing"}"""::Nil)
rdd:org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[35] at parallelize at <console>:22
scala> sqlContext.jsonRDD(rdd)
res19:org.apache.spark.sql.DataFrame = [address:string,name:string]
scala> sqlContext.jsonRDD(rdd).isLocal
res20:Boolean = false
```

十二、printSchema

1. 定义

```
def printSchema():Unit
```

2. 功能描述

以树型结构将 DataFrame 的 Schema 信息打印到控制台上。

3. 示例

```
scala> df.printSchema
root
 |-- age:long (nullable = true)
 |-- deptId:long (nullable = true)
 |-- gender:string (nullable = true)
 |-- job number:string (nullable = true)
```

```
| -- name:string (nullable = true)
| -- salary:long (nullable = true)
```

树节点由列名及其数据类型组成，其中 nullable 表示该列是否可以取 null 值。



十三、registerTempTable

1. 定义

```
def registerTempTable(tableName:String):Unit
```

2. 功能描述

将 DataFrame 注册为指定名字的临时表。

3. 示例

```
scala > df.registerTempTable("people")
scala > val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <=
19")
15/04/04 08:31:39 INFO parse.ParseDriver:Parsing command:SELECT name FROM people WHERE
age >= 13 AND age <= 19
15/04/04 08:31:40 INFO parse.ParseDriver:Parse Completed
teenagers:org.apache.spark.sql.DataFrame = [name:string]
scala > teenagers.show
name
Justin
```

注册成临时表之后，可以使用 SQLContext 的 sql 方法，执行 SQL 语句。

十四、schema

1. 定义

```
def schema:StructType
```

2. 功能描述

返回 DataFrame 的 Schema 信息，对应类型为 StructType。

3. 示例

```
scala > df.schema
res29:org.apache.spark.sql.types.StructType = StructType(StructField(age,LongType,true),Struct-
Field(deptId,LongType,true),StructField(gender,StringType,true),StructField(job number,String-
Type,true),StructField(name,StringType,true),StructField(salary,LongType,true))
```

十五、toDF

1. 定义

```
def toDF(colNames:String*):DataFrame
def toDF():DataFrame
```

2. 功能描述

不带参数的 toDF 返回它本身，带字符串数组的参数时，返回新的 DataFrame，该 DataFrame 重命名了各列名。

3. 示例

```
scala > df. toDF("age","deptId","gender","jobId","name","salary")
res35:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,jobId:string,name:
string,salary:bigint]
scala > df. toDF("age","deptId","gender","job number","name","salary")
res33:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
scala > df. toDF("age","deptId","gender","job number","name")
java. lang. IllegalArgumentException:requirement failed:The number of columns doesn't match.
Old column names:age,deptId,gender,job number,name,salary
New column names:age,deptId,gender,job number,name
at scala. Predef$. require(Predef. scala:233)
scala > df. toDF
res36:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
```

注意: 在调用带参的 toDF 方法时，参数个数必须和调用者 DataFrame 的列个数一样。

十六、agg

1. 定义

```
def agg(expr:Column,exprs:Column *):DataFrame
def agg(exprs:Map[String,String]):DataFrame
def agg(exprs:Map[String,String]):DataFrame
def agg(aggExpr:(String,String),aggExprs:(String,String) *):DataFrame
```

2. 功能描述

agg 这一系列的方法，为 DataFrame 提供数据列不需要经过 groups 就可以执行的统计操作。

3. 示例

```
scala > df. agg(max($"age"),avg($"salary"))
res37:org. apache. spark. sql. DataFrame = [MAX( age):bigint,AVG( salary):double]
scala > df. agg(max($"age"),avg($"salary")) . show
MAX( age) AVG( salary)
33          4666. 666666666667
scala > df. groupBy(). agg(max($"age"),avg($"salary"))
res39:org. apache. spark. sql. DataFrame = [MAX( age):bigint,AVG( salary):double]
scala > df. groupBy(). agg(max($"age"),avg($"salary")). show
MAX( age) AVG( salary)
33          4666. 666666666667
```

可以看到，直接用 agg 方法和先用 groupBy 分组再调用 agg 方法的结果是一样的。这里分别统计了 age 的最大值和 salary 的平均值。

```
scala >   df. agg(Map("age" -> "min","salary" -> "mean"))
res21:org. apache. spark. sql. DataFrame = [MIN(age#86L):bigint,AVG(salary#91L):double]
scala >   df. agg(Map("age" -> "min","salary" -> "mean")). show
```

```
15/04/05 09:34:50 INFOmapred.FileInputFormat:Total input paths to process:1
MIN(age#86L) AVG(salary#91L)
19              4666.666666666667
```

这是使用 Map 作为参数的示例，分别统计列 age 的最小值和 salary 的平均值。



```
scala >   df.agg(("age" -> "min"),("salary" -> "mean"))
res24:org.apache.spark.sql.DataFrame = [MIN(age#86L):bigint,AVG(salary#91L):double]
scala >   df.agg(("age" -> "min"),("salary" -> "mean")).show
15/04/05 09:35:52 INFOmapred.FileInputFormat:Total input paths to process:1
MIN(age#86L) AVG(salary#91L)
19              4666.666666666667
```

这是使用二元组重复参数作为参数的示例，分别统计列 age 的最小值和 salary 的平均值。

十七、apply

1. 定义

```
def apply(colName:String):Column
def col(colName:String):Column
```

2. 功能描述

这两个方法都可以根据指定列名返回 DataFrame 的列，其类型为 Column。

3. 示例

```
scala > df("age")
res4:org.apache.spark.sql.Column = age
scala > df.col("age")
res9:org.apache.spark.sql.Column = age
```

十八、as

1. 定义

```
def as(alias:Symbol):DataFrame
def as(alias:String):DataFrame
```

2. 功能描述

调用 as 方法后，使用别名构建 DataFrame。

3. 解析

为了分析这个方法的作用，查看带 as 方法和不带的两种情况。

首先修改调试日志的级别，方便查看调试信息：

```
scala > Logger.getLogger("org.apache.spark.sql").setLevel(Level.DEBUG)
```

不带 as 方法时的调试信息：

```
scala > val sss = sqlContext.sql("select foo,bar from pokes").explain(true)
15/04/07 01:13:05 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local
no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting
```

```
to a remote metastore.
15/04/07 01:13:05 INFO parse.ParseDriver:Parsing command:select foo,bar from pokes
15/04/07 01:13:05 INFO parse.ParseDriver:Parse Completed
15/04/07 01:13:05 INFO metastore.HiveMetaStore:0:get_table:db=default tbl=pokes
15/04/07 01:13:05 INFO HiveMetaStore.audit:ugi=harli ip=unknown-ip-addrcmd=get_table:db
=default tbl=pokes
15/04/07 01:13:05 INFO metastore.HiveMetaStore:0:get_table:db=default tbl=pokes
15/04/07 01:13:05 INFO HiveMetaStore.audit:ugi=harli ip=unknown-ip-addrcmd=get_table:db
=default tbl=pokes
==Parsed Logical Plan==
'Project ['foo',bar]
'UnresolvedRelation [pokes],None
==Analyzed Logical Plan==
Project [foo#79,bar#80]
MetastoreRelation default,pokes,None
==Optimized Logical Plan==
MetastoreRelation default,pokes,None
==Physical Plan==
HiveTableScan [foo#79,bar#80],(MetastoreRelation default,pokes,None),None
Code Generation:false
==RDD==
sss:Unit=()
```

带 as 方法时的调试信息：

```
scala>val sss=sqlContext.sql("select foo,bar from pokes").as("alise").explain(true)
15/04/07 01:13:18 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local
no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting
to a remote metastore.
15/04/07 01:13:18 INFO parse.ParseDriver:Parsing command:select foo,bar from pokes
15/04/07 01:13:18 INFO parse.ParseDriver:Parse Completed
15/04/07 01:13:18 INFO metastore.HiveMetaStore:0:get_table:db=default tbl=pokes
15/04/07 01:13:18 INFO HiveMetaStore.audit:ugi=harli ip=unknown-ip-addrcmd=get_table:db
=default tbl=pokes
15/04/07 01:13:19 INFO metastore.HiveMetaStore:0:get_table:db=default tbl=pokes
15/04/07 01:13:19 INFO HiveMetaStore.audit:ugi=harli ip=unknown-ip-addrcmd=get_table:db
=default tbl=pokes
15/04/07 01:13:19 INFO metastore.HiveMetaStore:0:get_table:db=default tbl=pokes
15/04/07 01:13:19 INFO HiveMetaStore.audit:ugi=harli ip=unknown-ip-addrcmd=get_table:db
=default tbl=pokes
==Parsed Logical Plan==
'Subquery alise
'Project ['foo',bar]
 'UnresolvedRelation [pokes],None
==Analyzed Logical Plan==
Project [foo#86,bar#87]
MetastoreRelation default,pokes,None
==Optimized Logical Plan==
```



```
MetastoreRelation default,pokes,None
 == Physical Plan ==
HiveTableScan [foo#86,bar#87],(MetastoreRelation default,pokes,None),None
Code Generation:false
 == RDD ==
sss:Unit = ()
```

调用 as 方法后，仅在解析逻辑计划时，解析的第一步使用了别名：Subquery alias（在 Parsed Logical Plan 处）。

十九、distinct

1. 定义

```
def distinct:DataFrame
```

2. 功能描述

返回对 DataFrame 的 Rows 去重后的 DataFrame。

3. 示例

```
scala > val newPeople = sqlContext. jsonFile( " hdfs: //wxx214: 9000/user/harli/test/newPeople. json" )
15/04/03 12: 18: 30 INFOmapred. FileInputFormat:Total input paths to process:1
newPeople: org. apache. spark. sql. DataFrame = [ age: bigint, deptId: bigint, gender: string, job number:
string,name:string,salary:bigint]
scala > newPeople. show
15/04/03 12: 19: 16 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
32   1         male       007             John       4000
20   2         female     008             Herry      5000
26   3         male       009             Jack       6000
scala > val unionPeople = df. unionAll( newPeople). select( " name" ). distinct. show
15/04/04 09: 05: 04 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/04 09: 05: 04 INFOmapred. FileInputFormat:Total input paths to process:1
name
Justin
Jack
John
Andy
Michael
Herry
unionPeople:Unit = ()
```

示例中加载 newPeople. json 文件，构建了 newPeople（是个 DataFrame），通过 unionAll 方法合并 df 与 newPeople，然后选择有重复 Rows 的“name”列，最后调用 distinct 方法进行去重。

二十、except

1. 定义

```
def except(other:DataFrame):DataFrame
```

2. 功能描述

返回 DataFrame，包含当前 Frame 的 Rows，同时这些 Rows 不在另一个 Frame 中。相当于两个 DataFrame 做减法。

3. 示例

```
scala > df. except( nDf)
res13:org. apache. spark. sql. DataFrame = [ age:bigint, deptId:bigint, gender:string, job number:string, name:string, salary:bigint]
scala > df. except( nDf). show
15/04/04 09:08:34 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/04 09:08:34 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name      salary
33  1     male    001        Michael  3000
26  3     male    006        Jack     3000
20  2     female  005        Herry    7000
19  3     male    003        Justin   5000
30  2     female  002        Andy     4000
32  1     male    004        John     6000
```

二十一、explode

1. 定义

```
def explode[A, B]( inputColumn:String, outputColumn:String) (f: (A) => TraversableOnce[B]) ( implicit arg0:scala. reflect. api. JavaUniverse. TypeTag[B]) :DataFrame
def explode[A <: Product]( input: Column * ) (f: ( Row) => TraversableOnce[A]) ( implicit arg0: scala. reflect. api. JavaUniverse. TypeTag[A]) :DataFrame
```

2. 功能描述

返回一个新的 DataFrame，其中原来的每一列都被指定的函数扩展成零行或多行。

3. 示例

```
scala > case class Book( title:String, words:String)
defined class Book
scala > val df = sc. textFile( "/user/harli/test/book. txt"). map( _. split( ",")). map( b => Book (b(0), b(1))). toDF
df:org. apache. spark. sql. DataFrame = [title:string, words:string]
scala > case class Word( word:String)
defined class Word
scala > import org. apache. spark. sql. _
import org. apache. spark. sql. _
scala > val allWords = df. explode( words) { case Row( words:String) => words. split( " "). map( Word (_)) }
allWords:org. apache. spark. sql. DataFrame = [title:string, words:string, word:string]
scala > allWords. map{ row => "0 = " + row(0) + "; 1 = " + row(1) + "; 2 = " + row(2)}
. foreach( println)
0 = book_3; 1 = word_3 word_4; 2 = word_3
0 = book_3; 1 = word_3 word_4; 2 = word_4
0 = book_1; 1 = word_1 word_2 word_3; 2 = word_1
0 = book_1; 1 = word_1 word_2 word_3; 2 = word_2
```

```
0 = book_1; 1 = word_1 word_2 word_3; 2 = word_3
0 = book_2; 1 = word_2 word_3; 2 = word_2
0 = book_2; 1 = word_2 word_3; 2 = word_3
scala > val bookCountPerWord = allWords. groupBy("word"). agg(countDistinct("title")). count
bookCountPerWord: Long = 4
scala > val bookCountPerWord = allWords. groupBy("word"). agg(countDistinct("title")). show
COUNT(DISTINCT title)
1
2
3
1
bookCountPerWord: Unit = ()
```



二十二、filter

1. 定义

```
def filter(conditionExpr: String): DataFrame
def filter(condition: Column): DataFrame
def where(condition: Column): DataFrame
```

2. 功能描述

按参数指定的 SQL 表达式的条件过滤 DataFrame。

3. 示例

```
scala > df. filter("age  > 20")
res15: org. apache. spark. sql. DataFrame = [age: bigint, deptId: bigint, gender: string, job number: string,
name: string, salary: bigint]
scala > df. filter("age  > 20"). show
15/04/04 09:22:28 INFOmapred. FileInputFormat: Total input paths to process: 1
agedeptId gender job number name       salary
33   1       male     001          Michael   3000
30   2       female   002          Andy      4000
32   1       male     004          John      6000
26   3       male     006          Jack      3000
scala > df. where($"age"  > 25 && $"gender" === "male"). show
15/04/03 17:01:31 INFOmapred. FileInputFormat: Total input paths to process: 1
agedeptId gender job number name       salary
33   1       male     001          Michael   3000
32   1       male     004          John      6000
26   3       male     006          Jack      3000
```

二十三、groupBy

1. 定义

```
def groupByKey(): RDD[(K, Iterable[V])]
```

2. 功能描述

使用一个或多个指定的列对 DataFrame 进行分组，以便对它们执行聚合操作。

3. 示例

```
scala > df. groupBy( "gender" ). agg(
     | "age" -> "max",
     | "salary" -> "mean"
     | )
res3:org. apache. spark. sql. DataFrame = [gender:string, MAX(age#0L):bigint, AVG(salary#5L):
double]
```

示例中先根据“gender”列对 df 进行分组，分组后再求“age”的最大值和“salary”列的平均值。

二十四、intersect

1. 定义

```
def intersect(other:DataFrame):DataFrame
```

2. 功能描述

取两个 DataFrame 中同时存在的 Rows，返回 DataFrame。

3. 示例

```
scala > valnewPeople = sqlContext. jsonFile( "hdfs://wxx214:9000/user/harli/test/newPeople. json" )
15/04/03 12:18:30 INFOmapred. FileInputFormat:Total input paths to process:1
newPeople:org. apache. spark. sql. DataFrame = [age:bigint, deptId:bigint, gender:string, job number:
string, name:string, salary:bigint]
scala > df. select( "name" ). intersect( newPeople. select( $"name" ) ). show
15/04/03 15:20:11 INFOmapred. FileInputFormat:Totmal input paths to process:1
15/04/03 15:20:11 INFOmapred. FileInputFormat:Total input paths to process:1
name
Jack
John
Herry
```

二十五、join

1. 定义

```
def join(right:DataFrame, joinExprs:Column, joinType:String):DataFrame
def join(right:DataFrame, joinExprs:Column):DataFrame
def join(right:DataFrame):DataFrame
```

2. 功能描述

对两个 DataFrame 求 join 操作。不带参数时取笛卡儿积，仅带 join Exprs 时默认为 Inner Join，第三个 join 参数 joinType 可以指定具体的 join 操作。

3. 示例

1）与前面的案例一样，加载测试文件：

```
scala > val people = sqlContext. jsonFile( "hdfs://wxx214:9000/user/harli/test/people. json" )
15/04/03 11:03:01 INFOmapred. FileInputFormat:Total input paths to process:1
people:org. apache. spark. sql. DataFrame = [age:bigint, deptId:bigint, gender:string, job number:string,
name:string, salary:bigint]
scala > val dept  = sqlContext. load( "hdfs://wxx214:9000/user/harli/test/department. json", "json" )
```

```
15/04/03 11:03:01 INFOmapred.FileInputFormat:Total input paths to process:1
dept:org.apache.spark.sql.DataFrame = [deptId:bigint,name:string]
```

2）将部门信息和人员信息做外联操作：



```
scala > people.join(dept,people("deptId") === dept("deptId"),"outer").show
15/04/03 15:25:55 INFOmapred.;FileInputFormat:Total input paths to process:1
15/04/03 15:25:55 INFOmapred.FileInputFormat:Total input paths to process:1
agedeptId gender job number name     salary deptId name
33  1    male    001    Michael  3000   1    Development Dept
32  1    male    004    John     6000   1    Development Dept
20  2    female  005    Herry    7000   2    Personnel Dept
30  2    female  002    Andy     4000   2    Personnel Dept
19  3    male    003    Justin   5000   3    Testing Department
26  3    male    006    Jack     3000   3    Testing Department
```

二十六、limit

1. 定义

```
def limit(n:Int):DataFrame
```

2. 功能描述

返回 DataFrame 的前 n 个 Rows。

3. 示例

```
scala > df.limit(3)
res27:org.apache.spark.sql.DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
scala > df.limit(3).show
15/04/05 10:33:40 INFOmapred.FileInputFormat:Total input paths to process:1
agedeptId gender job number name     salary
33  1    male    001    Michael  3000
30  2    female  002    Andy     4000
19  3    male    003    Justin   5000
```

二十七、orderBy 和 sort

1. 定义

```
def orderBy(sortExprs:Column * ):DataFrame
def orderBy(sortCol:String,sortCols:String * ):DataFrame
def sort(sortExprs:Column * ):DataFrame
def sort(sortCol:String,sortCols:String * ):DataFrame
```

2. 功能描述

按指定的一列或多列进行排序，分别支持字符串或 Column 的参数列表。

3. 示例

```
scala > df.sort("job number").show(3)
15/04/03 11:13:35 INFOmapred.FileInputFormat:Total input paths to process:1
agedeptId gender job number name     salary
```

```
33  1   male    001     Michael 3000
30  2   female  002     Andy    4000
19  3   male    003     Justin  5000
scala > df. sort( $" job number". asc). show
15/04/03 11:13:57 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
33  1   male    001     Michael 3000
30  2   female  002     Andy    4000
19  3   male    003     Justin  5000
32  1   male    004     John    6000
20  2   female  005     Herry   7000
26  3   male    006     Jack    3000
scala > df. orderBy(" age" ," salary" )
res29:org. apache. spark. sql. DataFrame = [ age:bigint, deptId:bigint, gender:string, job number:string,
name:string, salary:bigint]
scala > df. orderBy(" age" ," salary" ). show
15/04/05 10:36:38 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
19  3   male    003     Justin  5000
20  2   female  005     Herry   7000
26  3   male    006     Jack    3000
30  2   female  002     Andy    4000
32  1   male    004     John    6000
33  1   male    001     Michael 3000
scala > df. orderBy( col(" age" ), df(" salary" ) ). show
15/04/05 10:36:53 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name    salary
19  3   male    003     Justin  5000
20  2   female  005     Herry   7000
26  3   male    006     Jack    3000
30  2   female  002     Andy    4000
32  1   male    004     John    6000
33  1   male    001     Michael 3000
```

二十八、sample

1. 定义

```
def sample( withReplacement:Boolean, fraction:Double) :DataFrame
def sample( withReplacement:Boolean, fraction:Double, seed:Long) :DataFrame
```

2. 功能描述

按指定因子对 DataFrame 的 Rows 进行取样，如果指定 withReplacement 为 true 时，使用指定的种子或随机的种子进行替换。

3. 示例

```
scala > df. sample( false, 0. 5)
res32:org. apache. spark. sql. DataFrame = [ age:bigint, deptId:bigint, gender:string, job number:string,
name:string, salary:bigint]
```



```
scala > df. sample( false,0. 5). show
15/04/05 10:41:32 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name salary
30  2    female  002      Andy   4000
32  1    male    004      John   6000
26  3    male    006      Jack   3000
scala > df. sample( false,0. 5,1). show
15/04/05 10:41:40 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name salary
30  2    female  002      Andy   4000
32  1    male    004      John   6000
26  3    male    006      Jack   3000
scala > df. sample( true,0. 5,1). show
15/04/05 10:41:49 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name      salary
33  1    male    001      Michael3000
19  3    male    003      Justin 5000
32  1    male    004      John   6000
```

当 withReplacement 为 false 时，指定的种子无效，为 true 时，会根据指定的种子，对应 Rows 的序号进行替换。

二十九、Select 系列

1. 定义

```
def select( col:String,cols:String * ):DataFrame
def select( cols:Column * ):DataFrame
def selectExpr( exprs:String * ):DataFrame
```

2. 功能描述

从 DataFrame 选取指定的列，返回 DataFrame。指定列有三种方式，可以用列名字符串的重复参数，或 Column 重复参数及列名表达式的多个参数来指定。

3. 示例

```
scala > df. select( "age" ,"name" ). show
15/04/05 10:46:55 INFOmapred. FileInputFormat:Total input paths to process:1
age name
33  Michael
30  Andy
19  Justin
32  John
20  Herry
26  Jack
scala > df. select( $"age" ,$"name" ). show
15/04/05 10:47:05 INFOmapred. FileInputFormat:Total input paths to process:1
age name
33  Michael
30  Andy
19  Justin
```

```
32   John
20   Herry
26   Jack
scala > df. select( col( " age" ) ,df( " name" ) ). show
15/04/05 10:47:20 INFOmapred. FileInputFormat:Total input paths to process:1
age name
33   Michael
30   Andy
19   Justin
32   John
20   Herry
26   Jack
scala > df. selectExpr( " age  + 1" ," name as newName" ," abs( salary)" ). show
15/04/05 10:49:27 INFOmapred. FileInputFormat:Total input paths to process:1
( age  + 1) newName Abs( salary)
34          Michael     3000
31          Andy        4000
20          Justin      5000
33          John        6000
21          Herry       7000
27          Jack        3000
```

三十、unionAll

1. 定义

```
def unionAll( other:DataFrame) :DataFrame
```

2. 功能描述

联合调用者和参数这两个 DataFrame 的 Rows。

3. 示例

```
//用 sqlContext. jsonFile 方法加载文件构建 newPeople
scala > val newPeople = sqlContext. jsonFile( " hdfs://wxx214:9000/user/harli/test/newPeople. json" )
15/04/03 12:18:30 INFOmapred. FileInputFormat:Total input paths to process:1
newPeople: org. apache. spark. sql. DataFrame = [ age: bigint, deptId: bigint, gender: string, job number:
string, name: string, salary: bigint]
//联合 df 和 newPeople
scala > df. unionAll( newPeople). show
15/04/03 12:18:32 INFOmapred. FileInputFormat:Total input paths to process:1
15/04/03 12:18:32 INFOmapred. FileInputFormat:Total input paths to process:1
agedeptId gender job number name      salary
33   1      male      001         Michael   3000
30   2      female    002         Andy      4000
19   3      male      003         Justin    5000
32   1      male      004         John      6000
20   2      female    005         Herry     7000
26   3      male      006         Jack      3000
32   1      male      007         John      4000
20   2      female    008         Herry     5000
```

```
26   3      male     009        Jack     6000
```

三十一、withColumn 和 withColumnRenamed

1. 定义



```
def withColumn(colName:String,col:Column):DataFrame
def withColumnRenamed(existingName:String,newName:String):DataFrame
```

2. 功能描述

对 DataFrame 列进行操作，withColumn 增加 DataFrame 的列信息，withColumnRenamed 则是对 DataFrame 的列进行重命名。

3. 示例

```
scala > df. withColumn("level",people("age") / 10). show
15/04/03 12:16:19 INFOmapred. FileInputFormat:Total i;
nput paths to process:1
agedeptId gender job number name     salary level
33   1      male     001        Michael  3000     3.3
30   2      female   002        Andy     4000     3.0
19   3      male     003        Justin   5000     1.9
32   1      male     004        John     6000     3.2
20   2      female   005        Herry    7000     2.0
26   3      male     006        Jack     3000     2.6
scala > val rnDept = df. withColumnRenamed("job numbe","jobId")
rnDept:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
```

三十二、insertInto、insertIntoJDBC 和 createJDBCTable

1. 定义

```
def insertInto(tableName:String):Unit
def insertInto(tableName:String,overwrite:Boolean):Unit
def insertIntoJDBC(url:String,table:String,overwrite:Boolean):Unit
def createJDBCTable(url:String,table:String,allowExisting:Boolean):Unit
```

2. 功能描述

insert 系列的方法：向指定表中增加 DataFrame 的 Rows 数据。带参数 overwrite 且为 true 时，insert into 会导致覆写原表的数据（即插入前先 truncate 表）。参数 url 用来指定数据库信息。

createJDBCTable 用于创建外部数据库的表，参数包含数据库连接的 url 信息，表名 table，以及 allowExisting 表示是否允许表已存在。如果 allowExisting 为 true，会在 create 表之前先 delete 表。

3. 示例

```
scala > val df = sqlContext. sql("select 1,3"). toDF("first","second")
15/04/07 00:31:37 WARN conf. HiveConf:DEPRECATED:Configuration property hive. metastore. local
no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting
```

```
to a remote metastore.
15/04/07 00:31:37 INFO parse.ParseDriver:Parsing command:select 1,3
15/04/07 00:31:37 INFO parse.ParseDriver:Parse Completed
df:org.apache.spark.sql.DataFrame = [first:int,second:int]
scala > df.registerTempTable("test")
scala > val dbtable = "TEST_JDBC"
dbtable:String = TEST_JDBC
scala > df.createJDBCTable("jdbc:mysql://192.168.242.131:3306/hive? user = root&password = mysql",dbtable ,true)
scala > val idf = sqlContext.sql("select 2,4").toDF("first","second")
15/04/07 00:31:39 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/04/07 00:31:39 INFO parse.ParseDriver:Parsing command:select 2,4
15/04/07 00:31:39 INFO parse.ParseDriver:Parse Completed
idf:org.apache.spark.sql.DataFrame = [first:int,second:int]
scala > idf.insertIntoJDBC("jdbc: mysql://192.168.242.131: 3306/hive? user = root&password = mysql",dbtable ,false)
scala > val jdbcDF = sqlContext.load("jdbc",Map(
     |     "url" -> "jdbc: mysql: //192.168.242.131:3306/hive? user = root&password = mysql",
     |     "dbtable" -> dbtable))
jdbcDF:org.apache.spark.sql.DataFrame = [first:int,second:int]
scala > jdbcDF.show
first second
1     3
2     4
```

示例中首先构建了两个 DataFrame，用其中一个调用 createJDBCTable 方法构建了一个表：“TEST_JDBC”，这里 createJDBCTable 的第三个参数设置为 true，当表存在时会先 drop 表，然后再 create 表。

创建表之后，使用 insertIntoJDBC 方法，将第二个 DataFrame 插入到刚创建的表“TEST_JDBC”中，其中，insertIntoJDBC 的第三个参数选择了 false，因此不会覆盖原有的表数据。

三十三、save

1. 定义

```
def save(source:String,mode:SaveMode,options:Map[String,String]):Unit
def save(path:String,source:String,mode:SaveMode):Unit
def save(path:String,source:String):Unit
def save(path:String,mode:SaveMode):Unit
def save(path:String):Unit
```

2. 功能描述

将 DataFrame 的数据保存到指定路径下，其中 path 为数据存储路径，source 为数据源标识，mode 为保存模型，各个模型的具体信息可以参见章节 3.3.1 通用的加载/保存功能的案例与解析部分的保存模型的内容。

3. 示例

```
scala > case class Book( title:String,words:String)
defined class Book
scala > val df = sc. textFile("/user/harli/test/book. txt"). map(_. split(",")). map(b => Book (b(0),b(1))). toDF
df:org. apache. spark. sql. DataFrame = [title:string,words:string]
scala > case class Word( word:String)
defined class Word
scala > import org. apache. spark. sql. _
import org. apache. spark. sql. _
scala > val allWords = df. explode( words) { case Row(words:String) => words. split(" "). map(Word(_)) }
allWords:org. apache. spark. sql. DataFrame = [title:string,words:string,word:string]
scala > allWords. save("/user/harli/allword. json","json")
15/04/07 12:21:59 INFO output. FileOutputCommitter:Saved output of task' attempt_201504071221_0018_m_000000_620' to hdfs://wxx214:9000/user/harli/allword. json/_temporary/0/task_201504071221_0018_m_000000
15/04/07 12:21:59 INFO output. FileOutputCommitter:Saved output of task' attempt_201504071221_0018_m_000001_621' to hdfs://wxx214:9000/user/harli/allword. json/_temporary/0/task_201504071221_0018_m_000001
```

通过 hdfs 命令查看/user/harli/allword. json:

```
[harli@ wxx215 hadoop -2. 6. 0]$. /bin/hdfs dfs -ls /user/harli/allword. json
15/04/07 12:22:27 WARN util. NativeCodeLoader:Unable to load native -hadoop library for your platform... using builtin -java classes where applicable
Found 3 items
-rw-r--r--   3harli supergroup          0 2015-04-07 12:22 /user/harli/allword. json/_SUCCESS
-rw-r--r--   3harli supergroup        316 2015-04-07 12:22 /user/harli/allword. json/part-00000
-rw-r--r--   3harli supergroup        118 2015-04-07 12:22 /user/harli/allword. json/part-00001
```

三十四、saveAsParquetFile

1. 定义

```
def saveAsParquetFile(path:String):Unit
```

2. 功能描述

将 DataFrame 保存到数据源为“parquet”的指定路径下。

3. 示例

```
scala > case class Book( title:String,words:String)
defined class Book
scala > val df = sc. textFile("/user/harli/test/book. txt"). map(_. split(",")). map(b => Book (b(0),b(1))). toDF
df:org. apache. spark. sql. DataFrame = [title:string,words:string]
scala > case class Word( word:String)
```

```
defined class Word
scala > import org. apache. spark. sql. _
import org. apache. spark. sql. _
scala > val allWords = df. explode( words) { case Row(words:String) => words. split(" "). map(Word
(_)) }
allWords:org. apache. spark. sql. DataFrame = [title:string,words:string,word:string]
scala > allWords. save("/user/harli/allword. parquet","parquet")
```

通过 hdfs 命令查看路径"/user/harli/allword. parquet":

```
[harli@ wxx215 hadoop -2. 6. 0]$. /bin/hdfs dfs  -ls /user/harli/allword. parquet
15/04/07 12:28:06 WARN util. NativeCodeLoader:Unable to load native - hadoop library for your plat-
form... using builtin - java classes where applicable
Found 5 items
-rw-r--r--   3harli supergroup          0 2015-04-07 12:26 /user/harli/allword. parquet/
_SUCCESS
-rw-r--r--   3harli supergroup        365 2015-04-07 12:26 /user/harli/allword. parquet/_
common_metadata
-rw-r--r--   3harli supergroup        865 2015-04-07 12:26 /user/harli/allword. parquet/_
metadata
-rw-r--r--   3harli supergroup        922 2015-04-07 12:26 /user/harli/allword. parquet/
part-r-00001. parquet
-rw-r--r--   3harli supergroup        852 2015-04-07 12:26 /user/harli/allword. parquet/
part-r-00002. parquet
```

三十五、saveAsTable

1. 定义

```
def saveAsTable(tableName:String,source:String,mode:SaveMode,options:Map[String,String]):Unit
def saveAsTable(tableName:String,source:String,mode:SaveMode,options:Map[String,String]):Unit
def saveAsTable(tableName:String,source:String,mode:SaveMode):Unit
def saveAsTable(tableName:String,source:String):Unit
def saveAsTable(tableName:String,mode:SaveMode):Unit
def saveAsTable(tableName:String):Unit
```

2. 功能描述

将 DataFrame 保存到表中，参数和 save 方法一样。

3. 示例

```
scala > case class Book(title:String,words:String)
defined class Book
scala > val df = sc. textFile("/user/harli/test/book. txt"). map(_. split(",")). map(b => Book (b
(0),b(1))). toDF
df:org. apache. spark. sql. DataFrame = [title:string,words:string]
scala > df. saveAsTable("table","json")
15/04/07 12:32:02 INFOmetastore. HiveMetaStore:0:get_database:default
15/04/07 12:32:02 INFOHiveMetaStore. audit:ugi = harli    ip = unknown - ip - addrcmd = get_data-
base:default
15/04/07 12:32:02 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = table
15/04/07 12:32:02 INFOHiveMetaStore. audit:ugi = harli    ip = unknown - ip - addrcmd = get_table:db
= default tbl = table
```

```
15/04/07 12:32:02 INFOmapred.FileInputFormat:Total input paths to process:1
15/04/07 12:32:02 INFO output.FileOutputCommitter:Saved output of task' attempt_201504071232_0020_m_000001_625' to hdfs://wxx214:9000/user/hive/warehouse/table/_temporary/0/task_201504071232_0020_m_000001
15/04/07 12:32:02 INFO output.FileOutputCommitter:Saved output of task' attempt_201504071232_0020_m_000000_624' to hdfs://wxx214:9000/user/hive/warehouse/table/_temporary/0/task_201504071232_0020_m_000000
15/04/07 12:32:03 INFOmetastore.HiveMetaStore:0:create_table:Table(tableName:table,dbName:default,owner:harli,createTime:1428381123,lastAccessTime:0,retention:0,sd:StorageDescriptor(cols:[FieldSchema(name:col,type:array<string>,comment:from deserializer)],location:null,inputFormat:org.apache.hadoop.mapred.SequenceFileInputFormat,outputFormat:org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat,compressed:false,numBuckets:-1,serdeInfo:SerDeInfo(name:null,serializationLib:org.apache.hadoop.hive.serde2.MetadataTypedColumnsetSerDe,parameters:{serialization.format=1,path=hdfs://wxx214:9000/user/hive/warehouse/table}),bucketCols:[],sortCols:[],parameters:{},skewedInfo:SkewedInfo(skewedColNames:[],skewedColValues:[],skewedColValueLocationMaps:{})),partitionKeys:[],parameters:{spark.sql.sources.schema.part.0={"type":"struct","fields":[{"name":"title","type":"string","nullable":true,"metadata":{}},{"name":"words","type":"string","nullable":true,"metadata":{}}]},EXTERNAL=FALSE,spark.sql.sources.schema.numParts=1,spark.sql.sources.provider=json},viewOriginalText:null,viewExpandedText:null,tableType:MANAGED_TABLE)
15/04/07 12:32:03 INFOHiveMetaStore.audit:ugi=harli	ip=unknown-ip-addrcmd=create_table:Table(tableName:table,dbName:default,owner:harli,createTime:1428381123,lastAccessTime:0,retention:0,sd:StorageDescriptor(cols:[FieldSchema(name:col,type:array<string>,comment:from deserializer)],location:null,inputFormat:org.apache.hadoop.mapred.SequenceFileInputFormat,outputFormat:org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat,compressed:false,numBuckets:-1,serdeInfo:SerDeInfo(name:null,serializationLib:org.apache.hadoop.hive.serde2.MetadataTypedColumnsetSerDe,parameters:{serialization.format=1,path=hdfs://wxx214:9000/user/hive/warehouse/table}),bucketCols:[],sortCols:[],parameters:{},skewedInfo:SkewedInfo(skewedColNames:[],skewedColValues:[],skewedColValueLocationMaps:{})),partitionKeys:[],parameters:{spark.sql.sources.schema.part.0={"type":"struct","fields":[{"name":"title","type":"string","nullable":true,"metadata":{}},{"name":"words","type":"string","nullable":true,"metadata":{}}]},EXTERNAL=FALSE,spark.sql.sources.schema.numParts=1,spark.sql.sources.provider=json},viewOriginalText:null,viewExpandedText:null,tableType:MANAGED_TABLE)
15/04/07 12:32:03 INFO hive.log:Updating table stats fast for table
15/04/07 12:32:03 INFO hive.log:Updated size of table table to 136
15/04/07 12:32:03 INFOmetastore.HiveMetaStore:0:get_table:db=default tbl=table
15/04/07 12:32:03 INFOHiveMetaStore.audit:ugi=harli	ip=unknown-ip-addrcmd=get_table:db=default tbl=table
scala>sqlContext.table("table")
15/04/07 12:32:14 INFOmetastore.HiveMetaStore:0:get_table:db=default tbl=table
15/04/07 12:32:14 INFOHiveMetaStore.audit:ugi=harli	ip=unknown-ip-addrcmd=get_table:db=default tbl=table
res11:org.apache.spark.sql.DataFrame=[title:string,words:string]
```

三十六、flatMap

1. 定义

```
def flatMap[R](f:(Row)=>TraversableOnce[R])(implicit arg0:ClassTag[R]):RDD[R]
```

2. 功能描述

对 DataFrame 中 Rows 进行处理，并且将处理结果。

3. 示例

```
scala > sdf. flatMap( x => List( x(0) ,x(1) ) )
res14:org. apache. spark. rdd. RDD[ Any] = MapPartitionsRDD[20] at flatMap at DataFrame. scala:783
scala > sdf. flatMap( x => List( x(0) ,x(1) ) ). collect
res16:Array[ Any] = Array(1,2)
```

示例中将 Row 转换为由每一列组成的 List。

三十七、foreach

1. 定义

```
def foreach( f:( Row) => Unit) :Unit
def foreachPartition( f:( Iterator[ Row] ) => Unit) :Unit
```

2. 功能描述

foreach 方法上对 DataFrame 中的 Rows 进行处理。foreachPartition 方法则是对应分区中的 Rows 进行处理，即 Iterator ［Row］，使用方法类似。

3. 示例

```
scala > val sdf = sqlContext. sql( " select 1,2 " )
15/04/06 09:00:21 WARN conf. HiveConf:DEPRECATED:Configuration property hive. metastore. local no longer has any effect.  Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/04/06 09:00:21 INFO parse. ParseDriver:Parsing command:select 1,2
15/04/06 09:00:21 INFO parse. ParseDriver:Parse Completed
sdf:org. apache. spark. sql. DataFrame = [ _c0:int, _c1:int]
scala > sdf. foreach( x => println( " First = "  + x(0)  + " ; Second = "  + x(1) ) )
```

由于这是分布式计算，因此需要到 Executor 所在节点查看输出信息，查看 Web Interface 界面（http://master:8080），获取输出信息，依次跳转界面，如图 3.8 所示。

cluster01:8080

Spark 1.3.0 Spark Master at spark://cluster01:7077

URL: spark://cluster01:7077
REST URL: spark://cluster01:6066 *(cluster mode)*
Workers: 1
Cores: 4 Total, 4 Used
Memory: 6.0 GB Total, 512.0 MB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20150828080350-cluster01-44917	cluster01:44917	ALIVE	4 (4 Used)	6.0 GB (512.0 MB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150828080607-0000	Spark shell	4	512.0 MB	2015/08/28 08:06:07	harli	RUNNING	3.9 min

图 3.8　Spark 监控界面上的 Application 信息

监控界面上的特定 Executor 的日志信息如图 3.9 所示。



图 3.9　Spark 监控界面上特定 Executor 的日志信息

单击 Logs 下的 stdout，可以查看到输出信息，如图 3.10 所示。

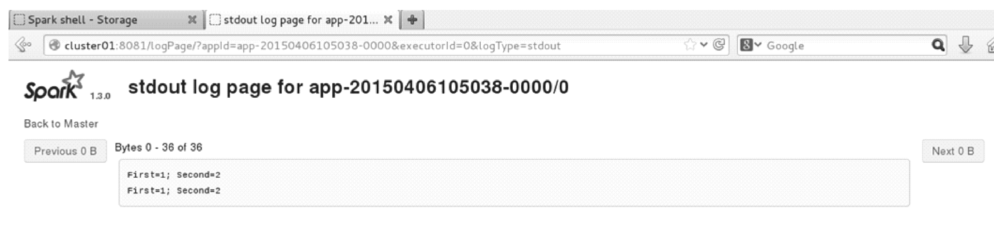


图 3.10　Spark 监控界面上特定 Executor 的 stdout 日志信息

由于当前执行了两次，因此 stdout 上有两行输出信息。

注意：这是在另一个集群中运行，cluster01 是运行 Spark 的 Master 进程的节点。

三十八、map 和 mapPartitions

1. 定义

```
def map[R](f:(Row) => R)(implicit arg0:ClassTaog[R]):RDD[R]
def mapPartitions[R](f:(Iterator[Row]) => Iterator[R])(implicit arg0:ClassTag[R]):RDD[R]
```

2. 功能描述

map 方法将 DataFrame 的 Row 按指定的函数参数映射成 R 实例，并返回以 R 为元素类型的 RDD 实例。

mapPartitions 方法和 map 类似，只是函数参数作用在 Iterator [Row]。

3. 示例

```
scala > val sdf = sqlContext.sql("select 1,2 ")
15/04/06 09:00:21 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local
no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting
to a remote metastore.
15/04/06 09:00:21 INFO parse.ParseDriver:Parsing command:select 1,2
15/04/06 09:00:21 INFO parse.ParseDriver:Parse Completed
sdf:org.apache.spark.sql.DataFrame = [_c0:int,_c1:int]
scala > sdf.map(x => "First = " + x(0) + "; Second = " + x(1))
```

```
res9:org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[56] at map at DataFrame.scala:776
scala > sdf.map(x => "First = " + x(0) + "; Second = " + x(1)).collect().foreach(println)
First = 1; Second = 2
```

4. 应用场景

这里重点分析 mapPartitions 的应用场景，该 API 对大数据量进行处理时，如果应用得当，可以极大提高计算性能，通过查看源码，可以看到许多性能优化都是通过直接调用该方法来实现的。

具体的应用场景，比如求 TopN 型的场景，如果基于大数据量进行排序然后取 topN，这在性能上是不可接受的，参看源码中使用了 mapPartitions 方法的 takeOrdered 方法：

```
def takeOrdered(num:Int)(implicit ord:Ordering[T]):Array[T] = {
  if (num == 0) {
    Array.empty
  } else {
    //用 mapPartitions 方法直接对 mapRDDs 的各个分区进行排序
    val mapRDDs = mapPartitions { items =>
      // Priority keeps the largest elements,so let s reverse the ordering.
      val queue = new BoundedPriorityQueue[T](num)(ord.reverse)
      queue ++= util.collection.Utils.takeOrdered(items,num)(ord)
Iterator.single(queue)
    }
    if (mapRDDs.partitions.size == 0) {
      Array.empty
    } else {
mapRDDs.reduce { (queue1,queue2) =>
        queue1 ++= queue2
        queue1
      }.toArray.sorted(ord)
    }
  }
}
```

可以看到，该方法将大数据量的聚合转变成了分区小数据量的聚合操作，这里的聚合是分区的 topN 操作，原理上，和 aggregate 型的 API 是一样的，只是在对分区聚合结果上的处理有点差异，aggregate 型的 API 是对分区的聚合结果再次进行二次聚合，然后封装成 RDD 类型返回，而 takeOrdered 方法，只需要对分区的聚合结果进行二次聚合，也就是上面源码中的 mapRDDs.reduce { (queue1,queue2)…部分，二次聚合后直接把得到的新的 topN 数组返回即可，不需要再封装成 RDD。

从上面的分析可以看到，对分布式的计算，基本原则就是化整为零，然后根据具体应用场景，对细节进行优化。因此，当需要实现类似于 TopN 的场景时，可以借鉴 takeOrdered 方法，但后面的具体细节处理，可以根据应用场景进行优化。

需要注意的是，takeOrdered 方法在分区聚合结果的处理上，是基于 N 的值比较小的情况下，如果分区数为 K 的话，那么 mapRDDs.reduce { (queue1,queue2)…} 这一步得到的数据集的大小就是 K * N（在每个分区都有 N 个的情况下），如果 K * N 数据量太大，超出内

存装载能力，那就可能出现 OMM 的问题了。解决方法还是一样的，就是根据前面讲的，根据具体场景的实际情况，在细节的处理上进行优化。比如：

1）先用 aggregate 型的 API 方法进行初步聚合，然后在得到的结果 RDD 上，重分区，合并各个分区的 TopN 值，这样 K 就减少成了 M，最终 Driver Program 端再进行 sort 时，数据集就可以从 K * N 变成了 M * N，可以有效避免内存不足（Out of Memory）的问题。其中减少分区的方法可以使用 coalesce 方法进行重分区，参数 shuffle 设置为 false，避免 shuffle 的过程。



2）也可以用另一种方法，修改 mapRDDs. reduce {（queue1，queue2）…} 方法的聚合操作，不再使用“queue1 ++= queue2”，而是将后面的 toArray. sorted(ord) 移入 reduce 中，每次 reduce 后取 queue1、queue2 的 TopN，这种方法其实就是再次利用 val mapRDDs = mapPartitions {…}. 中对分区排序的处理方式。实际上这种方法是没有必要的，如果数据量小，直接合并后再排序效率会更高，如果数据量大，那么在 Driver Program 端进行排序就失去了分布式并行计算的优势了。

这里使用“queue1 ++= queue2”是由于大部分场景下，TopN 的 N 值都不会太大，即使“++”也不会造成 OOM 的问题，而上面两种解决方法，都是一种用性能换内存的权衡结果，大部分场景下是不需要的。

三十九、repartition

1. 定义

```
def repartition(numPartitions:Int):DataFrame
```

2. 功能描述

返回一个 DataFrame，该 DataFrame 按指定 numPartitioins 对原 DataFrame 进行重分区。

3. 示例

```
scala > df. repartition(1). rdd. partitions. size
res13:Int = 1
scala > df. rdd. partitions. size
res16:Int = 2
```

示例中，重分区后返回的 DataFrame 对应的 RDD 的分区个数已经改为 1。

DataFrame 的分区，实际上对应其 RDD 的数据分区。因此分区个数也对应了 RDD 的分区个数。

四十、toJSON

1. 定义

```
def toJSON:RDD[String]
```

2. 功能描述

把 DataFrame 的内容用包含 JSON 字符串的 RDD 返回。

3. 示例

```
//调用 toJSON 进行转换
scala > df. toJSON
```

```
res4: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[18] at mapPartitions at DataFrame.scala:790
//用 collect 获取转换结果
scala > df.toJSON.collect
15/04/05 11:04:43 INFOmapred.FileInputFormat:Total input paths to process:1
res5:Array[String] = Array({"age":33,"deptId":1,"gender":"male","job number":"001","name":"Michael","salary":3000},{"age":30,"deptId":2,"gender":"female","job number":"002","name":"Andy","salary":4000},{"age":19,"deptId":3,"gender":"male","job number":"003","name":"Justin","salary":5000},{"age":32,"deptId":1,"gender":"male","job number":"004","name":"John","salary":6000},{"age":20,"deptId":2,"gender":"female","job number":"005","name":"Herry","salary":7000},{"age":26,"deptId":3,"gender":"male","job number":"006","name":"Jack","salary":3000})
```

四十一、queryExecution

1. 定义

```
val queryExecution:QueryExecution
```

2. 功能描述

返回 DataFrame 的查询执行语句，包含逻辑计划和物理计划。

3. 示例

```
scala > df.queryExecution
res44:org.apache.spark.sql.SQLContext#QueryExecution =
== Parsed Logical Plan ==
Relation[age#86L,deptId#87L,gender#88,job number#89,name#90,salary#91L] JSONRelation(/user/harli/data/people.json,1.0,None)
== Analyzed Logical Plan ==
Relation[age#86L,deptId#87L,gender#88,job number#89,name#90,salary#91L] JSONRelation(/user/harli/data/people.json,1.0,None)
== Optimized Logical Plan ==
Relation[age#86L,deptId#87L,gender#88,job number#89,name#90,salary#91L] JSONRelation(/user/harli/data/people.json,1.0,None)
== Physical Plan ==
PhysicalRDD[age#86L,deptId#87L,gender#88,job number#89,name#90,salary#91L],MapPartitionsRDD[135] at map at JsonRDD.scala:41
Code Generation:false
== RDD ==
```

Section 3.3 Spark SQL 处理各种数据源的案例与解析

通常在企业中使用的数据有多种格式，因此需要支持多种数据来源的处理，将不同数据源集成到企业统一的大数据平台下。Spark SQL 支持从各种数据源加载文件构建 DataFrame，以及将 DataFrame 保存到各种数据源中。

在进行数据源集成之前，先对 Spark SQL 的数据源进行分析，下面是从源码角度，对内

置的数据源、数据源的查找两个方面进行分析。

查看源码，可以从任何一个加载数据源的接口触发，最后找到解析数据源的代码。这里数据源的源码在 ddl. scala 文件中，相关代码如下所示。

1. 构建数据源的源码

```
private val builtinSources = Map(
  "jdbc"  -> classOf[org.apache.spark.sql.jdbc.DefaultSource],
  "json"  -> classOf[org.apache.spark.sql.json.DefaultSource],
  "parquet"  -> classOf[org.apache.spark.sql.parquet.DefaultSource]
)
```

可以看到，这里 Spark SQL 内置的数据源支持缩写方式，包含“jdbc”“json”和“parquet”这三种，对应类如图 3. 11 所示。

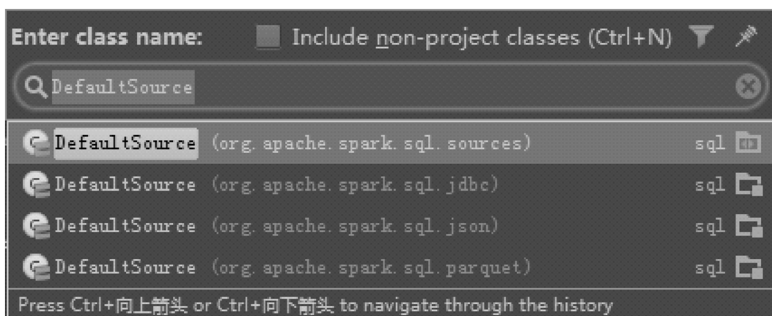


图 3. 11　Spark SQL 内置的数据源类

另外，通过数据源的查找的源码可以看出，查找时可以指定数据源类名的全路径的前缀。

2. 查找数据源的源码

```
/** 根据给定的名字,查找对应的数据源的类的定义. */
def lookupDataSource(provider:String):Class[_] = {
  //在内置数据源中,直接返回类名
  if (builtinSources.contains(provider)) {
    return builtinSources(provider)
  }
  //不在内置数据源中,加载给定名字指定的类
  val loader = Utils.getContextOrSparkClassLoader
  try {
    loader.loadClass(provider)
  } catch {
    case cnf:java.lang.ClassNotFoundException =>
      try {
        loader.loadClass(provider + ".DefaultSource")
      } catch {
```

```
                case cnf:java.lang.ClassNotFoundException =>
                  sys.error(s"Failed to load class for data source:$provider")
              }
          }
      }
```

这里可以看到，当查找数据源时，会从内置支持是三种的数据源中先进行查找，查找失败时，以输入的数据源类路径加类名“.DefaultSource”构建出数据源实例。

可以通过继承特质 RelationProvider 来自定义数据源类来扩展 Spark SQL，现有的继承类如图 3.12 所示。

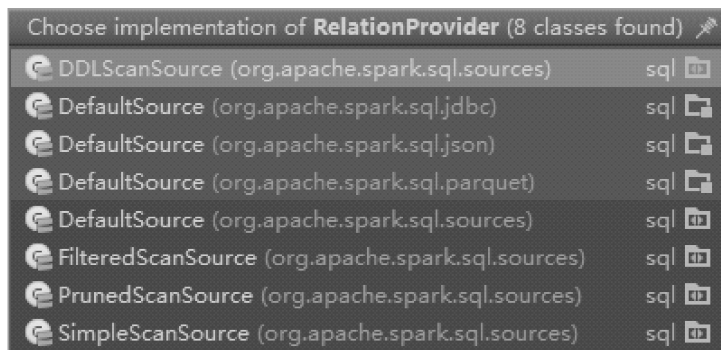


图 3.12　Spark SQL 数据源的 DefaultSource 类

3.3.1 通用的加载/保存功能的案例与解析

Spark SQL 提供了两种方式从各种数据源加载数据以构建 DataFrame，可以使用特定的方法或通用的方法以默认的数据源来直接加载数据源，也可以通过指定具体数据源的方法，用通用的方法来加载数据。

同时 Spark SQL 也提供 DataFrame 的持久化操作。

这部分内容介绍了如何加载和持久化 DataFrame。为了方便查看反馈信息，以交互式方式启动 Spark。

一、加载/保存数据源的最简单的方式

加载、保存 DataFrame 的最简单的方式是使用默认的数据源来进行所有操作。默认的数据源是内置的“parquet”数据源，可以通过修改配置 sparkSpark.sql.sources.default 将其他数据源作为默认值。

1. 使用针对特定数据源的方法

下边介绍 SQLContext 提供的针对特定数据源加载文件的方法，包括加载方法和保存方法。

1）加载数据源，代码如下所示：

```
scala> val people = sqlContext.jsonFile("/user/harli/data/people.json")
15/04/05 00:56:13 INFOmapred.FileInputFormat:Total input paths to process:1
```

```
people:org.apache.spark.sql.DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
scala > val people = sqlContext.parquetFile("/user/harli/data/people.parquet")
people:org.apache.spark.sql.DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
```

这里使用了两种方法，jsonFile 和 parquetFile，分别加载“json”和“parquet”的数据源。

2）保存 DataFrame 实例到数据源，代码如下所示：

```
scala > people.saveAsParquetFile("save.parquet")
```

查看 Web Interface 界面，如图 3.13 所示。

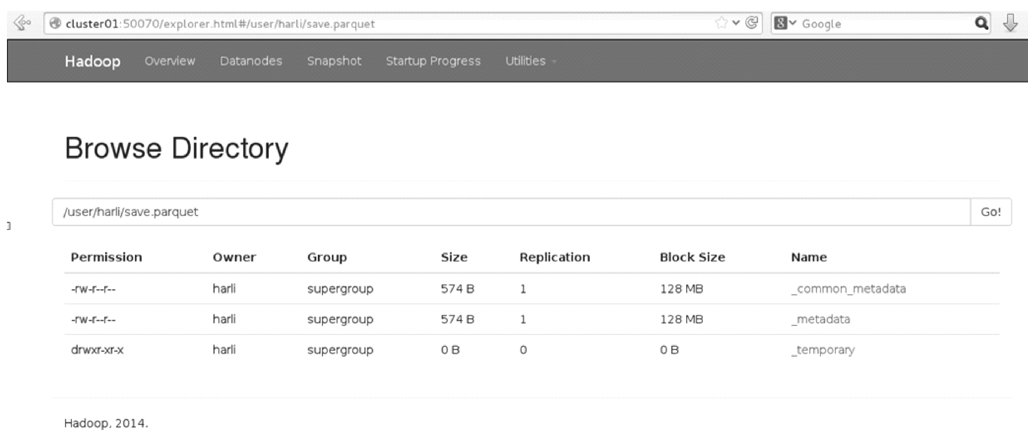


图 3.13　Hadoop 文件系统在 saveAsParquetFile 后的界面

当前没有指定具体路径，在使用 HDFS 作为存储系统时，默认会放在 HDFS 文件系统中当前用户的目录下，即/user/harli 目录，在指定的路径目录下，可以看到文件已经保存成功。

注意：如果目录以/开头，则对应的是 HDFS 的根目录（对应 core-site.xml 中的 *fs.defaultFS* 配置属性），而非当前用户所在目录下。比如/tmp，对应为 hdfs://namenode:port/tmp。其中 namenode 为启动 NameNode 进程的节点，当前环境下的节点地址为 192.168.70.214。

2. 使用默认数据源的方法

1）确认默认的数据源是否已经设置。可以通过查询当前的默认配置，确保是默认的“parquet”数据源，命令行输入如下：

```
scala > sc.getConf.get("spark.sql.sources.default","a")
res1:String = a
```

可以看到，当前没有设置默认的数据源，此时如果使用默认数据源去加载（或者保存）的话会报错。

通过 spark-shell 的 -conf 选项以 Key = Value 的形式来设置默认数据源参数，重新启动 spark-shell：

```
[harli@wxx215 spark-1.3.0-bin-hadoop2.4]$./bin/spark-shell --master spark://
192.168.70.214:7077 -conf "spark.sql.sources.default"="parquet"
```

这里使用 - conf 选项将配置属性 spark. sql. sources. default 设置为 parquet，进入交互式界面后，重新查询该属性值，语句如下：

```
scala > sc.getConf.get("spark.sql.sources.default","a")
res0:String = parquet
```

默认数据源已经成功设置。

2）加载数据源。以默认的数据源“parquet”加载文件：

```
//以通用加载数据源方法,来加载默认数据源的文件
scala > val people = sqlContext.load("/user/harli/data/people.parquet")
people:org.apache.spark.sql.DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
```

可以继续从加载后的 people 中选取某些列，然后保存到默认数据源上，如：

```
//以通用持久化方法,来持久化默认数据源的文件
scala > df.select("name","age").save("namesAndAges.parquet")
15/04/05 00:43:17 INFOhadoop.ParquetFileReader:Initiating action with parallelism:5
```

这里从 df 中选取了“name”，“age”两列信息，然后保存到 HDFS 文件系统中当前用户的目录下的“namesAndAges. parquet”目录中。

通过 Web Interface（http://namenode:50070）界面查看文件信息，如图 3.14 所示。

cluster01:50070/explorer.html#/user/harli/namesAndAges.parquet

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Browse Directory

/user/harli/namesAndAges.parquet Go!

Permission	Owner	Group	Size	Replication	Block Size	Name
-rw-r--r--	harli	supergroup	0 B	1	128 MB	_SUCCESS
-rw-r--r--	harli	supergroup	280 B	1	128 MB	_common_metadata
-rw-r--r--	harli	supergroup	592 B	1	128 MB	_metadata
-rw-r--r--	harli	supergroup	565 B	1	128 MB	part-r-00001.parquet
-rw-r--r--	harli	supergroup	537 B	1	128 MB	part-r-00002.parquet

图 3.14　hadoop 文件系统在 save 后的界面

然后将整个 people 保存为 parquet 格式，目录为 people. parquet：

```
scala > people.save("/user/harli/data/people.parquet")
15/04/05 00:34:31 INFOmapred.FileInputFormat:Total input paths to process:1
15/04/05 00:34:32 INFOhadoop.ParquetFileReader:Initiating action with parallelism:5
```

这里可以看到，输入文件的类型为 FileInputFormat，通过 hadoop. ParquetFileReader 读取，读取后的并行度（parallelism）为 5。

二、指定数据源的方式

1. 指定数据源加载文件

这里使用 load 方法，加载文件 people. json，对应数据源为“json”。

```
//以通用加载数据源的方法,指定具体数据源来加载文件
scala > val people = sqlContext. load("/user/harli/data/people. json","json")
15/04/05 00:31:43 INFOmapred. FileInputFormat:Total input paths to process:1
people:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
```

这里使用 load 方法，加载文件 people. parquet，对应数据源为“parquet”。

```
scala > val people = sqlContext. load("/user/harli/data/people. parquet","parquet")
people:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
```

2. 指定数据源保存文件

将加载后的 people 保存到 HDFS 文件系统当前用户的目录下的“save. json”目录中，保存为 json 文件：

```
scala > people. save("save. json","json")
```

通过 Web Interface(http://namenode:50070)界面查看文件信息，如图 3. 15 所示。

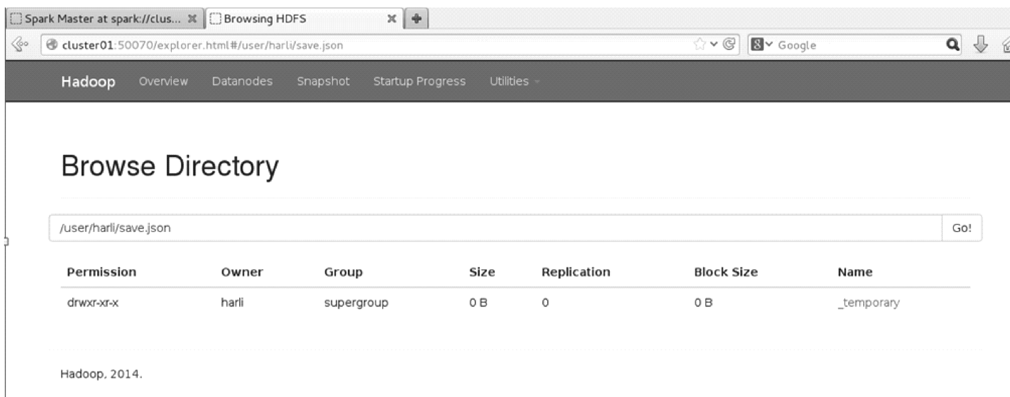


图 3. 15 Hadoop 文件系统在 save 后的界面

三、保存模式（Save Modes）

保存操作时可选择保存模式方式，来指定如果现有数据已经存在的话该如何处理。重要的是要意识到这些保存模式不使用任何锁操作，而且也不具备原子性。因此，当尝试对同一位置进行多个写操作时，写操作是不安全的。另外，当执行一个覆盖操作（SaveMode. Overwrite）时，在写新数据之前会先删除原有数据。具体的保存模型参考表 3. 2 的内容。

表 3.2 保存模式及其含义

Scala/Java	Python	含义
SaveMode. ErrorIfExists (default)	“error” (default)	当保存一个 DataFrame 到一个数据源时，如果数据已经存在，将会抛出一个异常
SaveMode. Append	“append”	当保存一个 DataFrame 到一个数据源时，如果数据/表已经存在，将会把 DataFrame 的数据添加到现有的数据中
SaveMode. Overwrite	“overwrite”	overwrite 模式意味着当保存一个 DataFrame 到一个数据源时，如果数据/表已经存在的话，将会用 DataFrame 的数据覆盖现有的数据
SaveMode. Ignore	“ignore”	ignore 模式意味着当保存一个 DataFrame 到一个数据源时，如果数据/表已经存在的话，将不会保存 DataFrame 的数据，也不会修改现有的数据。这与 SQL 中的“CREATE TABLE IF NOT EXISTS 操作类似

在实际保存操作中，需要注意各种数据源对保存模式使用的限制，比如“parquet”类型的数据源当前只支持 overwrite 的保存模式，当使用其他保存模式时会报不支持的错误。

四、保存到持久化的表中

先加载文件到 people 中：

```
scala > val people = sqlContext. load("/user/harli/data/people. json","json")
15/04/05 00:31:43 INFOmapred. FileInputFormat:Total input paths to process:1
people:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
```

使用 saveAsTable 方法，将 people 保存到表 people 中：

```
scala > people. saveAsTable("people")
15/04/05 01:39:25 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = people
15/04/05 01:39:25 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db =
default tbl = people
15/04/05 01:39:25 INFOmetastore. HiveMetaStore:0:get_database:default
15/04/05 01 : 39 : 25 INFOHiveMetaStore. audit: ugi = harliip = unknown - ip - addrcmd = get_
database:default
15/04/05 01:39:25 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = people
15/04/05 01:39:25 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db =
default tbl = people
```

持久化到表中和注册为临时表是不一样的，临时表在应用退出后会自动销毁，而持久化到表中是持久化到存储系统上，应用退出后不会销毁。

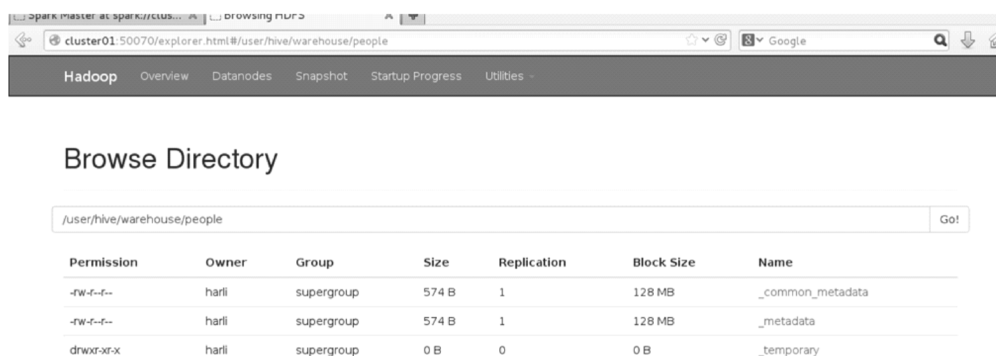
保存后的 Web Interface 界面，如图 3.16 所示。

如图所示，保存后自动创建了目录 hive 及其子目录 warehouse，并在该子目录下生成了保存表 people 的目录。

五、应用场景

1. Spark SQL 应用程序可以集成各种类型的数据源，包含不同数据源之间的存储转换、格式转换等，比如，将 json 文件格式转换为 parquet 格式，将 HDFS 上的 json 文件存储到 jdbc 中等。

2. 基于“one stack to rule them all”的思想，Spark 中的各个子框架和库之间可以实现无缝的数据共享和操作，而基于 Spark SQL 对各种数据源的支持，同时就是为其他各个子框



Permission	Owner	Group	Size	Replication	Block Size	Name
-rw-r--r--	harli	supergroup	574 B	1	128 MB	_common_metadata
-rw-r--r--	harli	supergroup	574 B	1	128 MB	_metadata
drwxr-xr-x	harli	supergroup	0 B	0	0 B	_temporary

图 3.16 Hadoop 文件系统在持久化到表后的界面

架，Spark Streaming、MLlib、GraphX 提供了各种数据源的支持。因此，在其他子框架需要时，可以使用 Spark SQL 来加载或持久化数据。

3.3.2 Parquet 文件处理的案例与解析

一、加载数据

通过加载 parquet 数据源，并将加载后的 people 注册到临时表中，然后使用 SQL 语句对该临时表进行操作，最后将操作结果打印出来。

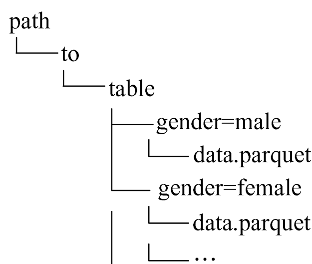
如下所示：

```
scala > val people = sqlContext. parquetFile("/user/harli/data/people. parquet")
people:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
scala > people. registerTempTable("parquetFile")
scala > val teenagers = sqlContext. sql("SELECT name FROM parquetFile WHERE age >= 13 AND age
<= 19")
15/04/05 01:50:36 INFO parse. ParseDriver: Parsing command: SELECT name FROM parquetFile
WHERE age >= 13 AND age <= 19
15/04/05 01:50:36 INFO parse. ParseDriver:Parse Completed
teenagers:org. apache. spark. sql. DataFrame = [name:string]
scala > teenagers. show
name
Justin
scala > teenagers. map(t => "Name:" + t(0)). collect(). foreach(println)
Name:Justin
```

二、分区发现

在类似于 Hive 的系统上，表分区是一种常见的优化方法。在一个分区表中，数据通常存储在不同的目录中，并将分区列值编码到每个分区目录的路径上。现在，Parquet 数据源可以自动地发现和推导分区信息。

例如，我们可以添加额外的列 “gender”，作为我们的分区列，用以下目录结构，可将 DataFrame 实践案例中的员工信息数据存储到分区表中。



通过将分区表的路径传递到 SQLContext. parquetFile 或 SQLContext. load，Spark SQL 会自动地从路径中提取分区信息。这时候返回的 DataFrame 的 schema 就构建成功了。

```
root
|-- name:string (nullable = true)
|-- age:long (nullable = true)
|-- gender:string (nullable = true)
|-- salary:string (nullable = true)
```

注意：分区表中用于分区的列的数据类型是自动推导的，目前自动推导支持数字数据类型和字符串类型。

具体的示例可以参考下一部分内容：合并 schema。

三、合并 schema

与 ProtocolBuffer，Avro 和 Thrift 一样，Parquet 也支持 schema 演变，用户可以先使用一个简单的 schema，然后根据需要逐步添加更多的列到 schema 中。通过这种方式，最终可以让多个不同的 Parquet 在 schema 上互相兼容。Parquet 数据源目前可以自动检测到这种情况，并合并所有这些文件。

以下案例说明了合并 schema 的详细步骤：

```
//构建一个元素值为(i,i*2)的 RDD,并转化为列名为("single","double")的 DataFrame
scala > val df1 = sc. makeRDD(1 to 5). map(i => (i,i * 2)). toDF("single","double")
df1:org. apache. spark. sql. DataFrame = [single:int,double:int]
//将构建的 df1 存储到 test_table 的 key =1 子目录下
scala > df1. saveAsParquetFile("data/test_table/key =1")
15/04/05 03:42:59 INFOhadoop. ParquetFileReader:Initiating action with parallelism:5
scala > val df2 = sc. makeRDD(6 to 10). map(i => (i,i * 3)). toDF("single","triple")
df2:org. apache. spark. sql. DataFrame = [single:int,triple:int]
//将构建的 df1 存储到 test_table 的 key =2 子目录下
scala > df2. saveAsParquetFile("data/test_table/key =2")
15/04/05 03:43:48 INFOhadoop. ParquetFileReader:Initiating action with parallelism:5
//解析包含前面两个子目录的 test_table 目录为 parquet 数据源
//并自动推导出对应的 schema 信息
scala > val df3 = sqlContext. parquetFile("data/test_table")
df3:org. apache. spark. sql. DataFrame = [single:int,double:int,triple:int,key:int]
scala > df3. printSchema()
root
|-- single:integer (nullable = true)
|-- double:integer (nullable = true)
```

```
|-- triple:integer (nullable = true)
|-- key:integer (nullable = true)
```

示例中，首先通过 SparkContext 的 makeRDD 方法构建一个 RDD 实例，然后调用 RDD 实例的 toDF 方法指定 schema，生成 DataFrame 实例，对应的 schema 信息为：[single: int,double:int]。然后将该 DataFrame 实例 df1 保存到分区路径“data/test_table/key = 1”下。

用相同的方法构建 DataFrame 实例 df2，对应的 schema 为[single:int,triple:int]，df2 的 schema 信息比 df1 添加列“triple”列，同时删除了“double”列，然后保存到分区路径“data/test_table/key =2”下。

最后读取分区表到 parquet，其路径为包含“data/test_table/key = 1”和“data/test_table/key =2”两个路径的 data/test_table 路径。加载构建 DataFrame 实例 df3 之后，会自动推导出 df3 的 schema 信息，最终推导出的 schema 信息为[single:int,double:int,triple:int,key:int]，包含列 df1 和 df2 的“single”，“double”和“triple”列信息，以及出现在分区路径上的“key”分区列信息。

3.3.3 JSON 数据集操作的案例与解析

Spark SQL 可以自动推导出一个 JSON 数据集的 schema 并加载它来构建一个 DataFrame。这种转换可以通过 SQLContext 的以下两种方法之一来实现。

1）jsonFile：可以从一个 JSON 文件的目录加载数据，文件中的每一行都对应一个 JSON 对象。

2）jsonRDD：可以从一个现有的 RDD 实例中加载数据，RDD 中的每一个元素都是包含一个 JSON 对象的字符串。

注意：作为 jsonFile 的文件不是一个典型的 JSON 文件。每一行必须包含一个单独的、有效的 JSON 对象。因此，普通的多行 JSON 文件通常会失败。

通过 jsonFile 加载文件构建 DataFrame 的示例：

```
scala > val people = sqlContext. jsonFile("/user/harli/data/people. json")
15/04/05 06:52:36 INFOmapred. FileInputFormat:Total input paths to process:1
people:org. apache. spark. sql. DataFrame = [age:bigint,deptId:bigint,gender:string,job number:string,
name:string,salary:bigint]
scala > people. printSchema
root
 |-- age:long (nullable = true)
 |--deptId:long (nullable = true)
 |-- gender:string (nullable = true)
 |-- job number:string (nullable = true)
 |-- name:string (nullable = true)
 |-- salary:long (nullable = true)
scala > people. registerTempTable("people")
scala > val teenagers = sqlContext. sql("SELECT name FROM people WHERE age >= 13 AND age <=
19")
```

```
15/04/05 06:54:52 INFO parse. ParseDriver:Parsing command:SELECT name FROM people WHERE
age >= 13 AND age <= 19
15/04/05 06:54:53 INFO parse. ParseDriver:Parse Completed
teenagers:org. apache. spark. sql. DataFrame = [name:string]
scala > teenagers. show
15/04/05 06:55:06 INFOmapred. FileInputFormat:Total input paths to process:1
name
Justin
```

示例中加载 people. json 这个文件构建出 people，并通过 printSchema 方法打印出 people 的自动推导出的 schema 信息，同时，将 people 注册为临时表“people”，然后通过 sql 方法用 SQL 查询语句从“people”临时表中查找年龄在指定范围的名字信息。

通过现有 RDD 构建 DataFrame 的示例：

```
//构建 List 实例,元素类型为 json 格式的字符串
scala > """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" ::Nil
res3:List[String] = List({"name":"Yin","address":{"city":"Columbus","state":"Ohio"}})
//构建 RDD 实例,元素类型为 json 格式的字符串
scala > valanotherPeopleRDD = sc. parallelize("""{"name":"Yin","address":{"city":"Colum-
bus","state":"Ohio"}}"""::Nil)
anotherPeopleRDD:org. apache. spark. rdd. RDD[String] = ParallelCollectionRDD[9] at parallelize at <
console >:22
//从元素类型为 json 格式的字符串的 RDD 构建出 DataFrame 实例
scala > val anotherPeople = sqlContext. jsonRDD(anotherPeopleRDD)
anotherPeople:org. apache. spark. sql. DataFrame = [address:struct < city:string, state:string >, name:
string]
scala > anotherPeople. printSchema
root
 |-- address:struct (nullable = true)
 |     |-- city:string (nullable = true)
 |     |-- state:string (nullable = true)
 |-- name:string (nullable = true)
```

示例中，通过“"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}"""::Nil”构建 Scala 数据集，类型为 List[String]。其中字符串“"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}"""”是 json 格式，包含两个成员“name”和“address”，其中“address”又由一个数据结构组成，包含“city”和“state”两个成员。

通过 SparkContext 的 parallelize 方法从 List[String]构建出 RDD 后，再通过 SQLContext 的 jsonRDD 方法将 RDD 实例转换为 DataFrame 实例。通过 printSchema 可以看到构建时自动推导出的 schema。

注册临时表，然后使用 sql 进行统计：

```
scala > anotherPeople. registerTempTable("people")
//以 name 列进行分组,并根据 count(个数统计值)进行排序
//然后查找 name 和 address. city 的个数统计
```

```
//注意:这里使用的是默认的 HiveContext,默认对应 hive 查询解析器
scala > val dd = sqlContext. sql( "select name,count( address. city) as count from people group by name order by count" )
15/04/23 04:38:16 WARNHiveConf: DEPRECATED: Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/04/23 04:38:16 INFOParseDriver: Parsing command: select name,count( address. city) as count from people group by name order by count
15/04/23 04:38:16 INFOParseDriver: Parse Completed
dd: org. apache. spark. sql. DataFrame = [ name: string, count: bigint]
scala > dd. show
name count
Yin   1
```

对临时表的 sql 操作，根据 name 进行 group by，以 address. city 统计个数进行 order by，然后获取 name 和 address. city 统计个数信息，构建新的 DataFrame。

Spark SQL 通过 Spark. sql. dialect 这个配置属性，来设置 Spark SQL 使用哪种解析器来解析查询的 SQL 语句，可以通过 SQLContext 实例的 setConf 方法或在 sql 方法中使用一个“SET key = value” 命令来设置该配置属性，对于 SQLContext，仅支持取值为“sql” 的 dialect，Spark SQL 使用取值为“sql” 的 dialect 来提供简单的 SQL 解析器。而对于 HiveContext，默认使用的 dialect 取值为“hiveql”，同时 dialect 的取值为“sql” 也是支持的。由于 HiveQL 解析器是更完整的，一般建议使用取值为“hiveql” 的 dialect。

```
//使用 getConf 方法获取当前的配置属性
scala > sqlContext. getConf( "spark. sql. dialect" , "" )
res8: String = ""
//在 sql 中使用 set 命令设置配置属性
scala > sqlContext. sql( "set spark. sql. dialect = sql" )
res10: org. apache. spark. sql. DataFrame = [ : string]
scala > sqlContext. getConf( "spark. sql. dialect" , "" )
res11: String = sql
scala > val dd = sqlContext. sql( "select name, count( address. city) as addcount from people group by name order by addcount" )
dd: org. apache. spark. sql. DataFrame = [ name: string, addcount: bigint]
scala > dd. show
nameaddcount
Yin   1
```

注意：在使用不同的 dialect 时，注意 SQL 的语法支持，比如在 dialect 为“sql” 时，不要使用保留字来命名，如：

```
scala > val dd = sqlContext. sql( "select name, count( address. city) as count from people group by name order by count" )
java. lang. RuntimeException: [1. 37] failure: `union' expected but `count' found
select name, count( address. city) as count from people group by name order by count
    at scala. sys. package$. error( package. scala:27)
    at org. apache. spark. sql. catalyst. AbstractSparkSQLParser. apply( AbstractSparkSQLParser. scala:40)
```

```
at org. apache. spark. sql. SQLContext$$anonfun$2. apply(SQLContext. scala:130)
at org. apache. spark. sql. SQLContext$$anonfun$2. apply(SQLContext. scala:130)
at org. apache. spark. sql. SparkSQLParser$$anonfun$org$apache$spark$sql$SparkSQLParser$$others$1. apply(SparkSQLParser. scala:96)
at org. apache. spark. sql. SparkSQLParser$$anonfun$org$apache$spark$sql$SparkSQLParser$$others$1. apply(SparkSQLParser. scala:95)
```

这里使用 count 这个保留字作为别名，从而导致了 SQL 查询解析器语法解析时出错，执行命令失败。修改别名后：

```
scala > sqlContext. sql("set spark. sql. dialect = sql")
res22:org. apache. spark. sql. DataFrame = [ :string]
scala > val dd = sqlContext. sql("select name,count(address. city) as count1 from people group by name order by count1")
dd:org. apache. spark. sql. DataFrame = [name:string,count1:bigint]
scala > dd. show
name count1
Yin    1
```

对应的，dialect 为“hiveql”时，同样的命令是可以正确执行的：

```
scala > sqlContext. sql("set spark. sql. dialect = hiveql")
res19:org. apache. spark. sql. DataFrame = [ :string]
scala > sqlContext. getConf("spark. sql. dialect","")
res20:String = hiveql
scala > val dd = sqlContext. sql("select name,count(address. city) as count from people group by name order by count")
15/04/23 05:11:33 WARNHiveConf: DEPRECATED: Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/04/23 05:11:33 INFOParseDriver:Parsing command:select name,count(address. city) as count from people group by name order by count
15/04/23 05:11:33 INFOParseDriver:Parse Completed
dd:org. apache. spark. sql. DataFrame = [name:string,count:bigint]
scala > dd. show
name count
Yin    1
```

注意：在使用 SQL 查询语句时，需要注意使用语句的语法正确性。查询当前的“spark. sql. dialect”值即可获取使用的具体查询解析器名称。

3. 3. 4 操作 Hive 表的案例与解析

Spark SQL 还支持读写存储在 Apache Hive 中的数据。Spark 1. 3 版本中，Spark 默认构建实例时包含“ – Phive”启用和“ – Phive – thriftserver”这两个 profile，因此实例构建后就已经支持 Hive。可以通过将现有的 Hive 部署的 hive – site. xml 配置文件放置到 Spark 部署的 conf 目录下，来访问现有的 Hive 数据仓库。

当使用 Hive 时必须构造一个 HiveContext 实例，HiveContext 继承自 SQLContext，其功能

是 SQLContext 功能的超集合，可以使用更加完备的 HiveQL 解析器来写查询语句，使用 Hive UDFs，并可以访问 Hive 的 MetaStore，从 Hive 表中读取数据。并且所有在 SQLContext 下可以获取的数据，在 HiveContext 中仍然可以获取。

我们不需要使用一个现有的 Hive 也可以创建一个 HiveContext 实例，当没有使用 hive-site. xml 配置文件时，context 会自动地在当前目录下创建 metastore_db 和 warehouse。

我们可以通过配置属性 spark. sql. dialect 来指定 SQL 所使用的查询解析器，具体使用方法可以参考下面的案例。

下面给出两种情况下使用 HiveQL 语句的示例。

一、不使用现有的 Hive 环境

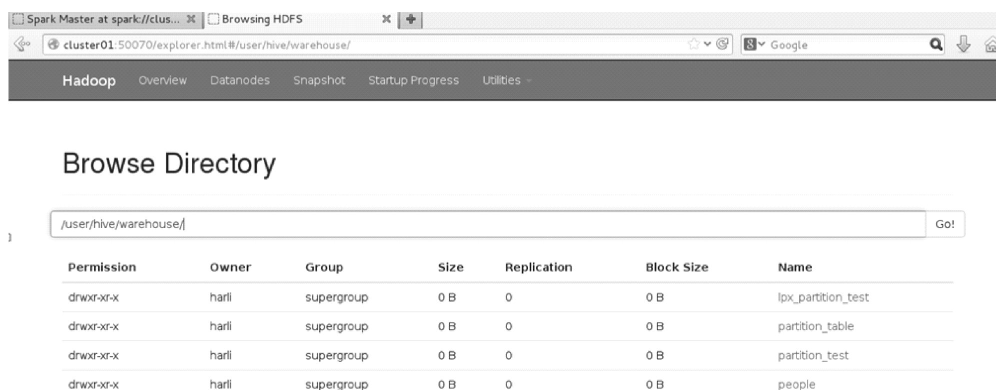
1. 构建表

通过“sqlContext. sql("CREATE TABLE IF NOT EXISTS src (key INT,value STRING)")”语句使用 HiveQL 的建表语句，构建了一个名为 src 的 Hive 表，包含 key 和 value 两列。

```
scala > sqlContext. sql("CREATE TABLE IF NOT EXISTS src (key INT,value STRING)")
15/04/05 07:38:08 INFO parse. ParseDriver:Parsing command:CREATE TABLE IF NOT EXISTS src
(key INT,value STRING)
15/04/05 07:38:08 INFO parse. ParseDriver:Parse Completed
15/04/05 07:38:08 INFO log. PerfLogger: <PERFLOG method = Driver. run from = org. apache. hadoop.
hive. ql. Driver >
15/04/05 07:38:08 INFO log. PerfLogger: <PERFLOG method = TimeToSubmit from = org. apache.
hadoop. hive. ql. Driver >
15/04/05 07:38:08 INFO ql. Driver:Concurrency mode is disabled,not creating a lock manager
15/04/05 07:38:08 INFO log. PerfLogger: <PERFLOG method = compile from = org. apache. hadoop.
hive. ql. Driver >
15/04/05 07:38:08 INFO log. PerfLogger: <PERFLOG method = parse from = org. apache. hadoop.
hive. ql. Driver >
15/04/05 07:38:08 INFO parse. ParseDriver:Parsing command:CREATE TABLE IF NOT EXISTS src
(key INT,value STRING)
15/04/05 07:38:08 INFO parse. ParseDriver:Parse Completed
15/04/05 07:38:08 INFO log. PerfLogger: </PERFLOG method = parse start = 1428244688097 end =
1428244688098 duration = 1 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:08 INFO log. PerfLogger: <PERFLOG method = semanticAnalyze from = org. apache.
hadoop. hive. ql. Driver >
15/04/05 07:38:08 INFO parse. SemanticAnalyzer:Starting Semantic Analysis
15/04/05 07:38:08 INFO parse. SemanticAnalyzer:Creating table src position = 27
15/04/05 07:38:08 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:08 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db =
default tbl = src
15/04/05 07:38:08 INFOmetastore. HiveMetaStore:0:get_database:default
15/04/05 07 : 38 : 08 INFOHiveMetaStore. audit: ugi = harliip = unknown - ip - addrcmd = get_
database:default
15/04/05 07:38:08 INFO ql. Driver:Semantic Analysis Completed
15/04/05 07 : 38 : 08 INFO log. PerfLogger: </PERFLOG method = semanticAnalyze start =
1428244688098 end = 1428244688101 duration = 3 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:08 INFO ql. Driver: Returning Hive schema: Schema(fieldSchemas: null, properties:
null)
```

```
15/04/05 07:38:08 INFO log.PerfLogger: </PERFLOG method=compile start=1428244688097 end=1428244688102 duration=5 from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO log.PerfLogger: <PERFLOG method=Driver.execute from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO ql.Driver: Starting command: CREATE TABLE IF NOT EXISTS src (key INT, value STRING)
15/04/05 07:38:08 INFO log.PerfLogger: </PERFLOG method=TimeToSubmit start=1428244688097 end=1428244688102 duration=5 from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO log.PerfLogger: <PERFLOG method=runTasks from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO log.PerfLogger: <PERFLOG method=task.DDL.Stage-0 from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO exec.DDLTask: Default to LazySimpleSerDe for table src
15/04/05 07:38:08 INFOmetastore.HiveMetaStore: 0: create_table: Table(tableName:src, dbName:default, owner:harli, createTime:1428244688, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:[FieldSchema(name:key, type:int, comment:null), FieldSchema(name:value, type:string, comment:null)], location:null, inputFormat:org.apache.hadoop.mapred.TextInputFormat, outputFormat:org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null, serializationLib:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, parameters:{serialization.format=1}), bucketCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNames:[], skewedColValues:[], skewedColValueLocationMaps:{}), storedAsSubDirectories:false), partitionKeys:[], parameters:{}, viewOriginalText:null, viewExpandedText:null, tableType:MANAGED_TABLE)
15/04/05 07:38:08 INFOHiveMetaStore.audit: ugi=harliip=unknown-ip-addrcmd=create_table: Table(tableName:src, dbName:default, owner:harli, createTime:1428244688, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:[FieldSchema(name:key, type:int, comment:null), FieldSchema(name:value, type:string, comment:null)], location:null, inputFormat:org.apache.hadoop.mapred.TextInputFormat, outputFormat:org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null, serializationLib:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, parameters:{serialization.format=1}), bucketCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNames:[], skewedColValues:[], skewedColValueLocationMaps:{}), storedAsSubDirectories:false), partitionKeys:[], parameters:{}, viewOriginalText:null, viewExpandedText:null, tableType:MANAGED_TABLE)
15/04/05 07:38:08 INFO log.PerfLogger: </PERFLOG method=runTasks start=1428244688102 end=1428244688172 duration=70 from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO log.PerfLogger: </PERFLOG method=Driver.execute start=1428244688102 end=1428244688172 duration=70 from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO ql.Driver: OK
15/04/05 07:38:08 INFO log.PerfLogger: <PERFLOG method=releaseLocks from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO log.PerfLogger: </PERFLOG method=releaseLocks start=1428244688172 end=1428244688172 duration=0 from=org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:08 INFO log.PerfLogger: </PERFLOG method=Driver.run start=1428244688097 end=1428244688172 duration=75 from=org.apache.hadoop.hive.ql.Driver>
res14: org.apache.spark.sql.DataFrame = [result: string]
```

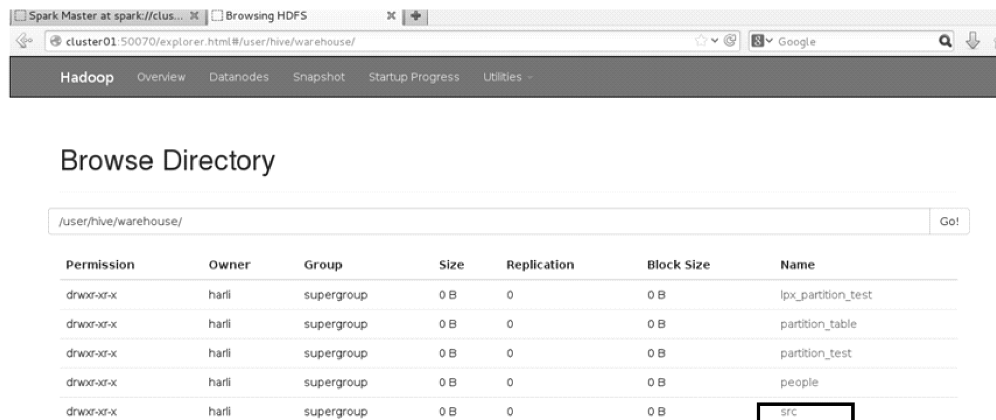
这是构建 src 表之前的 Web Interface 界面，如图 3.17 所示。



Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	harli	supergroup	0 B	0	0 B	lpx_partition_test
drwxr-xr-x	harli	supergroup	0 B	0	0 B	partition_table
drwxr-xr-x	harli	supergroup	0 B	0	0 B	partition_test
drwxr-xr-x	harli	supergroup	0 B	0	0 B	people

图 3. 17 Hadoop 文件系统在构建 src 表之前的界面

这是构建 src 表之后的 Web Interface 界面，如图 3. 18 所示。可以看到，src 表已经在 warehouse 目录下构建成功。



Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	harli	supergroup	0 B	0	0 B	lpx_partition_test
drwxr-xr-x	harli	supergroup	0 B	0	0 B	partition_table
drwxr-xr-x	harli	supergroup	0 B	0	0 B	partition_test
drwxr-xr-x	harli	supergroup	0 B	0	0 B	people
drwxr-xr-x	harli	supergroup	0 B	0	0 B	src

图 3. 18 Hadoop 文件系统在构建 src 表之后的界面

2. 加载本地文件到表中

通过 HiveQL 的 LOAD 语句“LOAD DATA LOCAL INPATH examples/src/main/resources/kv1. txt INTO TABLE src”，将本地文件系统中的 kv1. txt 文件加载到 src 表中。该文件位于 Spark 部署目录下。

```
scala > sqlContext. sql ( " LOAD DATA LOCAL INPATH examples/src/main/resources/kv1. txt INTO TABLE src" )
15/04/05 07:38:21 INFO parse. ParseDriver:Parsing command:LOAD DATA LOCAL INPATH examples/src/main/resources/kv1. txt INTO TABLE src
15/04/05 07:38:21 INFO parse. ParseDriver:Parse Completed
15/04/05 07:38:21 INFO log. PerfLogger: <PERFLOG method = Driver. run from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO log. PerfLogger: <PERFLOG method = TimeToSubmit from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO ql. Driver:Concurrency mode is disabled,not creating a lock manager
15/04/05 07:38:21 INFO log. PerfLogger: <PERFLOG method = compile from = org. apache. hadoop.
```

```
hive. ql. Driver >
15/04/05 07 : 38 : 21 INFO log. PerfLogger: < PERFLOG method = parse from =
org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO parse. ParseDriver:Parsing command:LOAD DATA LOCAL INPATH 'exam-
ples/src/main/resources/kv1. txt' INTO TABLE src
15/04/05 07:38:21 INFO parse. ParseDriver:Parse Completed
15/04/05 07:38:21 INFO log. PerfLogger: </PERFLOG method = parse start = 1428244701661 end =
1428244701662 duration = 1 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO log. PerfLogger: < PERFLOG method = semanticAnalyze from = org. apache.
hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:21 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db
= default tbl = src
15/04/05 07:38:21 INFO ql. Driver:Semantic Analysis Completed
15/04/05 07 : 38 : 21 INFO log. PerfLogger: </PERFLOG method = semanticAnalyze start =
1428244701662 end = 1428244701794 duration = 132 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO ql. Driver:Returning Hive schema:Schema(fieldSchemas:null,properties:
null)
15/04/05 07:38:21 INFO log. PerfLogger: </PERFLOG method = compile start = 1428244701661 end
= 1428244701794 duration = 133 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO log. PerfLogger: < PERFLOG method = Driver. execute from = org. apache.
hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO ql. Driver:Starting command:LOAD DATA LOCAL INPATH 'examples/src/
main/resources/kv1. txt' INTO TABLE src
15/04/05 07 : 38 : 21 INFO log. PerfLogger: </PERFLOG method = TimeToSubmit start =
1428244701661 end = 1428244701794 duration = 133 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO log. PerfLogger: < PERFLOG method = runTasks from = org. apache. hadoop.
hive. ql. Driver >
15/04/05 07 : 38 : 21 INFO log. PerfLogger: < PERFLOG method = task. COPY. Stage - 0 from =
org. apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO exec. Task:Copying data from file:/home/harli/cluster_13/spark/examples/
src/main/resources/kv1. txt to hdfs://cluster01:9000/tmp/hive - harli/hive_2015 - 04 - 05_07 - 38 -
21_661_2658555693652186087 - 1/ - ext - 10000
15/04/05 07:38:21 INFO exec. Task:Copying file:file:/home/harli/cluster_13/spark/examples/src/
main/resources/kv1. txt
15/04/05 07:38:21 INFO log. PerfLogger: < PERFLOG method = task. MOVE. Stage - 1 from = org.
apache. hadoop. hive. ql. Driver >
15/04/05 07:38:21 INFO exec. Task:Loading data to table default. src fromhdfs://cluster01:9000/
tmp/hive - harli/hive_2015 - 04 - 05_07 - 38 - 21_661_2658555693652186087 - 1/ - ext - 10000
15/04/05 07:38:21 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:21 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db
= default tbl = src
15/04/05 07:38:22 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:22 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db
= default tbl = src
15/04/05 07:38:22 INFO metadata. Hive:Renaming src:hdfs://cluster01:9000/tmp/hive - harli/hive
_2015 - 04 - 05_07 - 38 - 21_661_2658555693652186087 - 1/ - ext - 10000/kv1. txt;dest:hdfs://
cluster01:9000/user/hive/warehouse/src/kv1. txt;Status:true
15/04/05 07:38:22 INFOmetastore. HiveMetaStore:0:alter_table:db = default tbl = src newtbl = src
```

```
15/04/05 07:38:22 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = alter_table:db
 = default tbl = src newtbl = src
15/04/05 07:38:22 INFOmetastore.HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:22 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = get_table:db
 = default tbl = src
15/04/05 07:38:22 INFO hive.log:Updating table stats fast for src
15/04/05 07:38:22 INFO hive.log:Updated size of table src to 5812
15/04/05 07:38:22 INFO log.PerfLogger: <PERFLOG method = task.STATS.Stage - 2 from =
org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:22 INFO exec.StatsTask:Executing stats task
15/04/05 07:38:22 INFOmetastore.HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:22 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = get_table:db
 = default tbl = src
15/04/05 07:38:22 INFOmetastore.HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:22 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = get_table:db
 = default tbl = src
15/04/05 07:38:22 INFOmetastore.HiveMetaStore:0:alter_table:db = default tbl = src newtbl = src
15/04/05 07:38:22 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = alter_table:db
 = default tbl = src newtbl = src
15/04/05 07:38:22 INFOmetastore.HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:22 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = get_table:db
 = default tbl = src
15/04/05 07:38:22 INFO hive.log:Updating table stats fast for src
15/04/05 07:38:22 INFO hive.log:Updated size of table src to 5812
15/04/05 07:38:22 INFO exec.Task:Table default.src stats:[numFiles = 1,numRows = 0,totalSize =
5812,rawDataSize = 0]
15/04/05 07:38:22 INFO log.PerfLogger: </PERFLOG method = runTasks start = 1428244701795 end
 = 1428244702361 duration = 566 from = org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:22 INFO log.PerfLogger: </PERFLOG method = Driver.execute start =
1428244701794 end = 1428244702361 duration = 567 from = org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:22 INFO ql.Driver:OK
15/04/05 07:38:22 INFO log.PerfLogger: <PERFLOG method = releaseLocks from = org.apache.
hadoop.hive.ql.Driver>
15/04/05 07:38:22 INFO log.PerfLogger: </PERFLOG method = releaseLocks start = 1428244702361
end = 1428244702361 duration = 0 from = org.apache.hadoop.hive.ql.Driver>
15/04/05 07:38:22 INFO log.PerfLogger: </PERFLOG method = Driver.run start = 1428244701661
end = 1428244702361 duration = 700 from = org.apache.hadoop.hive.ql.Driver>
res15:org.apache.spark.sql.DataFrame = [result:string]
```

3. 查询 src 表

从 src 表中查询 key 和 value 字段。

```
scala>sqlContext.sql("FROM src SELECT key,value")
15/04/05 07:46:01 INFO parse.ParseDriver:Parsing command:FROM src SELECT key,value
15/04/05 07:46:01 INFO parse.ParseDriver:Parse Completed
15/04/05 07:46:01 INFOmetastore.HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:46:01 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = get_table:db =
default tbl = src
res17:org.apache.spark.sql.DataFrame = [key:int,value:string]
```

最后通过 HiveQL 的查询语句“FROM src SELECT key，value”，从 src 表中查询指定的 key 和 value 两列数据来构建出 DataFrame 实例。

4. 在界面输出 src 表的查询结果

获取查询结果并在终端显示。

```
scala > sqlContext. sql( "FROM src SELECT key,value" ). collect( ). foreach( println)
15/04/05 07:38:26 INFO parse. ParseDriver:Parsing command:FROM src SELECT key,value
15/04/05 07:38:26 INFO parse. ParseDriver:Parse Completed
15/04/05 07:38:26 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 07:38:26 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db =
default tbl = src
15/04/05 07:38:27 INFOmapred. FileInputFormat:Total input paths to process:1
[238,val_238]
[86,val_86]
[311,val_311]
[27,val_27]
[165,val_165]
[409,val_409]
[255,val_255]
[278,val_278]
……
```

5. 删除 src 表

使用 drop table 删除 src 表。

```
scala > sqlContext. sql( "DROP table src" )
15/04/05 08:42:04 INFO parse. ParseDriver:Parsing command:DROP table src
15/04/05 08:42:04 INFO parse. ParseDriver:Parse Completed
15/04/05 08:42:04 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 08:42:04 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db =
default tbl = src
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = Driver. run from = org. apache.
hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = TimeToSubmit from = org. apache.
hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO ql. Driver:Concurrency mode is disabled,not creating a lock manager
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = compile from = org. apache. hadoop.
hive. ql. Driver >
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = parse from = org. apache. hadoop.
hive. ql. Driver >
15/04/05 08:42:04 INFO parse. ParseDriver:Parsing command:DROP TABLE src
15/04/05 08:42:04 INFO parse. ParseDriver:Parse Completed
15/04/05 08:42:04 INFO log. PerfLogger: </PERFLOG method = parse start = 1428248524423 end =
1428248524423 duration =0 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = semanticAnalyze from = org. apache.
hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 08:42:04 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db =
default tbl = src
```

```
15/04/05 08:42:04 INFO ql. Driver:Semantic Analysis Completed
15/04/05 08 : 42 : 04 INFO log. PerfLogger: </PERFLOG method = semanticAnalyze start = 1428248524423 end = 1428248524447 duration = 24 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO ql. Driver:Returning Hive schema:Schema(fieldSchemas:null,properties:null)
15/04/05 08:42:04 INFO log. PerfLogger: </PERFLOG method = compile start = 1428248524423 end = 1428248524448 duration = 25 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = Driver. execute from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO ql. Driver:Starting command:DROP TABLE src
15/04/05 08:42:04 INFO log. PerfLogger: </PERFLOG method = TimeToSubmit start = 1428248524423 end = 1428248524448 duration = 25 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = runTasks from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08 : 42 : 04 INFO log. PerfLogger: < PERFLOG method = task. DDL. Stage - 0 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 08:42:04 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db = default tbl = src
15/04/05 08:42:04 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 08:42:04 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db = default tbl = src
15/04/05 08:42:04 INFOmetastore. HiveMetaStore:0:drop_table:db = default tbl = src
15/04/05 08:42:04 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = drop_table:db = default tbl = src
15/04/05 08:42:04 INFOmetastore. HiveMetaStore:0:get_table:db = default tbl = src
15/04/05 08:42:04 INFOHiveMetaStore. audit:ugi = harliip = unknown - ip - addrcmd = get_table:db = default tbl = src
15/04/05 08:42:04 INFOmetastore. hivemetastoressimpl:deleting  hdfs://cluster01:9000/user/hive/warehouse/src
15/04/05 08:42:04 INFO fs. TrashPolicyDefault:Namenode trash configuration:Deletion interval = 0 minutes,Emptier interval = 0 minutes.
15/04/05 08:42:04 INFOmetastore. hivemetastoressimpl:Deleted the diretoryhdfs://cluster01:9000/user/hive/warehouse/src
15/04/05 08:42:04 INFO log. PerfLogger: </PERFLOG method = runTasks start = 1428248524448 end = 1428248524675 duration = 227 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08 : 42 : 04 INFO log. PerfLogger: </PERFLOG method = Driver. execute start = 1428248524448 end = 1428248524675 duration = 227 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO ql. Driver:OK
15/04/05 08:42:04 INFO log. PerfLogger: < PERFLOG method = releaseLocks from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO log. PerfLogger: </PERFLOG method = releaseLocks start = 1428248524675 end = 1428248524675 duration = 0 from = org. apache. hadoop. hive. ql. Driver >
15/04/05 08:42:04 INFO log. PerfLogger: </PERFLOG method = Driver. run start = 1428248524423 end = 1428248524675 duration = 252 from = org. apache. hadoop. hive. ql. Driver >
res18:org. apache. spark. sql. DataFrame = [ ]
```

日志中包含了具体执行时的详细信息，比如 HiveMetaStore 部分，包含了当前使用的数据库，执行语句和表名等信息。在 metastore. hivemetastoressimpl 部分还包含了内部的实现细

节，比如由于 Hive 表不是外部表，这里删除表的同时，也删除了对应的 HDFS 存储系统上的文件。其他日志信息还包含执行时间、锁操作等。通过对日志信息的分析，可以了解内部的执行流程和细节。

此时查看 Web Interface 界面，可以看到 src 表已经被删除，如图 3. 19 所示。

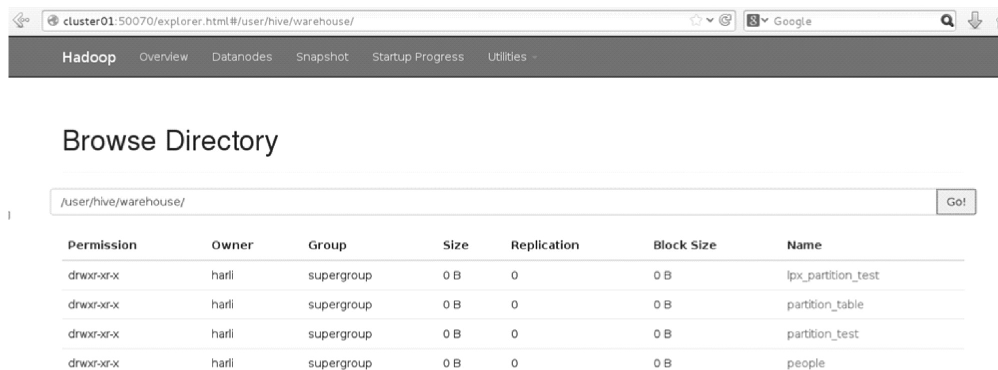


图 3. 19　Hadoop 文件系统在删除表之后的界面

二、使用现有的 Hive 数据仓库

（一）准备工作

1. 安装好 mysql 数据库。

2. 安装好 Hive 环境，当前 Hive 环境使用 MySQL 作为默认的元数据数据库。

1）已将 mysql – connector – java – 5. 1. 35 – bin. jar 复制到 hive/lib 目录下。

2）已用 Hive 的 jar 包替换了 Hadoop 较低版本的相同 jar 包：用 hive/lib/jline – 2. 12. jar 替换 share/hadoop/yarn/lib/jline – 0. 9. 94. jar ；如果没有替换，运行“. /bin/hive”时会报 jline 的类不兼容错误。

3）在 hive – env. sh 中按实际集群修改下面三个属性：

```
# Set HADOOP_HOME to point to a specifichadoop install directory
# HADOOP_HOME =${bin}/../../hadoop
export HADOOP_HOME = "/home/harli/cluster/hadoop"
export HIVE_HOME = "/home/harli/cluster/hive"
# Hive Configuration Directory can be controlled by:
# export HIVE_CONF_DIR =
export HIVE_CONF_DIR = "/home/harli/cluster/hive/conf"
```

4）hive – site. xml 配置示例：

```
<? xml – stylesheet type = "text/xsl" href = "configuration. xsl"? >
<configuration >
<property >
<name > hive. metastore. local </name >
<value > true </value >
</property >
<property >
<name > javax. jdo. option. ConnectionURL </name >
```

```
<value>jdbc:mysql://192.168.70.214:3306/hive?  =createDatabaseIfNotExist=true</value>
</property>
<property>
<name>javax.jdo.option.ConnectionDriverName</name>
<value>com.mysql.jdbc.Driver</value>
</property>
<property>
<name>javax.jdo.option.ConnectionUserName</name>
<value>root</value>
</property>
<property>
<name>javax.jdo.option.ConnectionPassword</name>
<value>mysql</value>
</property>
</configuration>
```



其中，javax. jdo. option. ConnectionURL 为数据库连接的 URL 信息，根据 MySQL 具体安装设置。当前连接的用户名和密码为 root/mysql，实际情况下最好不要使用 root 用户。

在此主要讲解 Spark SQL 的案例，因此只介绍最基本的 Hive 的环境配置。如果需要修改，可以参考 Hive 官网的部署文档。比如，可以增加 Hive 数据库的编码的属性设置：在配置 javax. jdo. option. ConnectionURL 中添加 "& characterEncoding = latin1"，metastore 的远程访问的配置中为 hive. metastore. uris 设置 value 为 thrift://cluster01:9083，等等。其中，character Encoding 设置的编码为 latin/。

注意：ConnectionURL 参数中的 & 符号在 XML 文件中需转义。驱动类比较旧时，比如 mysql－connector－java－5.1.6－bin.jar，在执行（比如 drop table）语句时可能会出现如下异常：

```
FAILED:Execution Error,return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask. MetaException
(message:javax.jdo.JDODataStoreException:You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use near' OPTION SQL_SELECT_
LIMIT=DEFAULT' at line 1
```

这是由于 MySQL 的版本和驱动的版本不匹配导致的，测试时使用的 MySQL 版本是 5.6，而案例使用的驱动是 mysql－connector－java－5.1.6－bin.jar，由于 MySQL 5.6 已经抛弃了这个参数，所以会报上面错误，这里换成最新的驱动 mysql－connector－java－5.1.35－bin.jar 后问题解决。

当前 MySQL 版本信息如下：

```
[harli@cluster01 hive]$mysql  -V
mysql   Ver 14.14 Distrib 5.6.23,for Linux (x86_64) usingEditLine wrapper
```

MySQL 驱动的下载地址：http://dev.mysql.com/downloads/connector/j/

（二）操作 Hive 表的案例

在 Hive 中启动 ./bin/hive

```
[harli@cluster01 hive]$./bin/hive
15/04/06 03:31:29 WARN conf.HiveConf:HiveConf of name hive.metastore.local does not exist
```

```
Logging initialized using configuration in file:/home/harli/cluster_13/hive/conf/hive-log4j.properties
SLF4J:Class path contains multiple SLF4J bindings.
SLF4J:Found binding in [jar:file:/home/harli/cluster_13/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J:Found binding in [jar:file:/home/harli/cluster_13/hive/lib/hive-jdbc-1.1.0-standalone.jar!!!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J:See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J:Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hive>
```

出现“hive>”提示后，开始创建一个表：

```
hive>show tables;
OK
Time taken:0.701 seconds
hive>CREATE TABLE pokes (foo INT,bar STRING);
OK
Time taken:0.523 seconds
hive>show tables;
OK
pokes
Time taken:0.036 seconds,Fetched:1 row(s)
```

查看 Web Interface 界面上的文件系统信息，如图 3.20 所示。

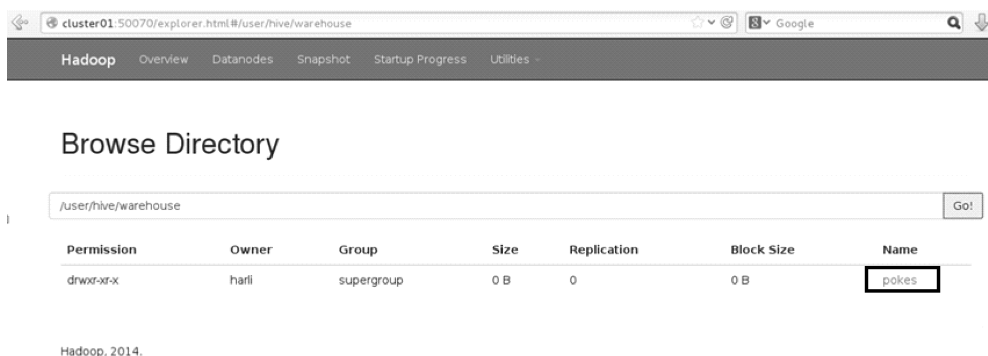


图 3.20 Hadoop 文件系统在 Hive 的 cli 中构建表之后的界面

可以看到，中/user/hive/warehouse 这个默认仓库目录下，生成列对应表名的目录 pokes。

将 mysql-connector-java-5.1.35-bin.jar 路径添加到 Spark_CLASSPATH 下，启动 Spark 脚本，脚本内容如下：

```
[harli@cluster01 spark]$Spark_CLASSPATH=$Spark_CLASSPATH:/home/harli/cluster/hive/lib/mysql-connector-java-5.1.35-bin.jar ./bin/spark-shell --master spark://cluster01:7077
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
15/04/06 03:38:21 WARN util.NativeCodeLoader:Unable to load native-hadoop library for your platform… using builtin-java classes where applicable
15/04/06 03:38:21 INFO spark.SecurityManager:Changing view acls to:harli
15/04/06 03:38:21 INFO spark.SecurityManager:Changing modify acls to:harli
```

```
15/04/06 03:38:21 INFO spark.SecurityManager:SecurityManager:authentication disabled; ui acls disabled; users with view permissions:Set(harli); users with modify permissions:Set(harli)
15/04/06 03:38:21 INFO spark.HttpServer:Starting HTTP Server
15/04/06 03:38:21 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/06 03:38:21 INFO server.AbstractConnector:Started SocketConnector@0.0.0.0:40779
15/04/06 03:38:21 INFO util.Utils:Successfully started service 'HTTP class server' on port 40779.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/
```

在交互式界面中查询现有的表信息：

```
scala> sqlContext.sql("show tables").show
15/04/06 03:57:47 INFOmetastore.HiveMetaStore:0:get_tables:db=default pat=.*
15/04/06 03:57:47 INFOHiveMetaStore.audit:ugi=harliip=unknown-ip-addrcmd=get_tables:db=default pat=.*
tableNameisTemporary
pokes        false
```

已经可以查看到 Hive 刚创建的表。

在 Spark SQL 中创建表：

```
scala> sqlContext.sql("CREATE TABLE pokes_test (foo INT,bar STRING)")
res0:org.apache.spark.sql.DataFrame=[result:string]
```

在 Hive 中查看：

```
hive> show tables;
OK
pokes
pokes_test
Time taken:0.683 seconds,Fetched:2 row(s)
```

注意： 如果启动 spark-shell 脚本时，没有将 mysql-connector-java-5.1.35-bin.jar 包添加到当前的 CLASS_PATH 路径下，在加载类时会找不到类导致 MySQL 的数据库连接失败，这时候会出现包含以下信息的异常提示：

```
at java.lang.reflect.Constructor.newInstance(Constructor.java:526)
at org.apache.hadoop.hive.metastore.MetaStoreUtils.newInstance(MetaStoreUtils.java:1410)
… 56 more
Caused by:javax.jdo.JDOFatalInternalException:Error creating transactional connection factory
NestedThrowables:
java.lang.reflect.InvocationTargetException
at org.datanucleus.api.jdo.NucleusJDOHelper.getJDOExceptionForNucleusException(NucleusJDOHelper.java:587)
at org.datanucleus.api.jdo.JDOPersistenceManagerFactory.freezeConfiguration(JDOPersistenceManagerFactory.java:788)
```

```
…

Caused by: org. datanucleus. store. rdbms. connectionpool. DatastoreDriverNotFoundException: The specified datastore driver ("com. mysql. jdbc. Driver") was not found in the CLASSPATH. Please check your CLASSPATH specification, and the name of the driver.
at org. datanucleus. store. rdbms. connectionpool. AbstractConnectionPoolFactory. loadDriver(AbstractConnectionPoolFactory. java:58)
```

其中，第一个 Caused by 处提示连接创建失败，继续往下查可以看到驱动类（com. mysql. jdbc. Driver）找不到的异常 DatastoreDriverNotFoundException。

小技巧：

1. 当日志出现 Error 时，跟踪其调用堆栈信息，查找关键字为 Caused by，可能其中一个异常堆栈信息中会出现多个 Caused by，而最下方的 Caused by，也就是最早抛出异常的地方，往往就是错误的根源，问题的根源才是解决问题的出发点。

2. 类找不到的根本原因，参考章节 3. 3. 5 使用 JDBC 操作其他数据库的案例与解析部分的故障排除的内容，主要是由 JVM 的类加载机制导致的。如果在当前执行环境（需要注意在分布式计算框架下，是对应真正执行的节点，与提交点等其他节点的环境无关）下的 CLASSPATH 中找不到该 jar 包，就会出现类找不到的异常。

3. Spark 计算框架已经提供了自动将所需 jar 包添加到执行环境的 CLASSPATH 上了，如果执行环境本身没有部署该 jar 包，就可以充分利用 Spark 的 --jars 参数来自动设置。

（三）远程访问 MetaStoreServer 的案例

修改 hive-site. xml 文件，添加：

```
<property>
<name>hive. metastore. uris</name>
<value>thrift://cluster01:9083</value>
</property>
```

远程访问 MetaStoreServer 时，需在服务器端启动 MetaStoreServer，客户端利用 Thrift 协议通过 MetaStoreServer 来访问元数据库。当没有启动 MetaStoreServer 服务时，执行 spark-shell 会出现如下错误信息：

```
scala> sqlContext. tableNames
15/04/11 03:53:01 WARNHiveConf: DEPRECATED: Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/04/11 03:53:01 WARNmetastore: Failed to connect to the MetaStore Server…
15/04/11 03:53:02 WARNmetastore: Failed to connect to the MetaStore Server…
15/04/11 03:53:03 WARNmetastore: Failed to connect to the MetaStore Server…
java. lang. RuntimeException: java. lang. RuntimeException: Unable to instantiate org. apache. hadoop. hive. metastore. HiveMetaStoreClient
    at org. apache. hadoop. hive. ql. session. SessionState. start(SessionState. java:346)
    at
……
Caused by: java. lang. RuntimeException: Unable to instantiate org. apache. hadoop. hive. metastore. HiveMetaStoreClient
```

```
at org.apache.hadoop.hive.metastore.MetaStoreUtils.newInstance(MetaStoreUtils.java:1412)
    at org.apache.hadoop.hive.metastore.RetryingMetaStoreClient.<init>(RetryingMetaStoreClient.java:62)
    at org.apache.hadoop.hive.metastore.RetryingMetaStoreClient.getProxy(RetryingMetaStoreClient.java:72)
    at org.apache.hadoop.hive.ql.metadata.Hive.createMetaStoreClient(Hive.java:2453)
    at org.apache.hadoop.hive.ql.metadata.Hive.getMSC(Hive.java:2465)
    at org.apache.hadoop.hive.ql.session.SessionState.start(SessionState.java:340)
    ... 56 more
Caused by:java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at
......
Caused by:MetaException(message:Could not connect to meta store using any of the URIs provided. Most recent failure:org.apache.thrift.transport.TTransportException:java.net.ConnectException:Connection refused
    at org.apache.thrift.transport.TSocket.open(TSocket.java:185)
......
Caused by:java.net.ConnectException:Connection refused
    at java.net.PlainSocketImpl.socketConnect(Native Method)
    at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:339)
    at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:200)
    at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:182)
    at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
    at java.net.Socket.connect(Socket.java:579)
    at org.apache.thrift.transport.TSocket.open(TSocket.java:180)
    ... 68 more
)
    at org.apache.hadoop.hive.metastore.HiveMetaStoreClient.open(HiveMetaStoreClient.java:382)
    at org.apache.hadoop.hive.metastore.HiveMetaStoreClient.<init>(HiveMetaStoreClient.java:214)
    ... 66 more
```

由于连接请求被拒绝导致 HiveMetaStoreClient 实例化失败，因此需要先启动远程的 MetaStoreServer 服务，启动代码如下：

```
#!/usr/bin/env bash
nohup ./bin/hive --service metastore >> metastore.log 2>&1 &
echo$! > hive-metastore.pid
```

这里以后台进程方式启动 metastore，同时保存进程 ID 到指定文件中。启动后，再次执行 spark-shell 后，运行命令访问 Hive 的 metastore 信息就可以正常执行了：

```
scala> sqlContext.tableNames
15/04/11 04:02:44 WARNHiveConf:DEPRECATED:Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
res0:Array[String] = Array(pokes,pokes_test)
```

三、扩展用户定义函数场景下的实践案例与解析

在 Spark 1.3 版本中，在 DataFrame 领域特定语言（DSL）或 SQL 中使用的 UDF，其注册操作被移到了 SQLContext 中。下面给出 UDF 注册实例，并基于该 UDF 的定义，给出 DataFrame 和 SQL 中的两种案例。

以下的数据准备和之前的案例是一样的：

```
scala > case class People(name:String,age:Int)
defined class People
scala > valrdd = sc.textFile("/user/harli/people.txt").map(_.split(",")).map(p => People(p(0),
p(1).trim.toInt))
rdd:org.apache.spark.rdd.RDD[People] = MapPartitionsRDD[30] at map at <console>:25
scala > val people = rdd.toDF()
people:org.apache.spark.sql.DataFrame = [name:string,age:int]
scala > people.registerTempTable("people")
```

其中 people.txt 是 Spark 提供的部署目录中的 resources 子目录下的文件。

注册名为 strLen 和 fmtNum 的两个 UDF：

```
//构建求字符串长度的 UDF
scala > sqlContext.udf.register("strLen",(s:String) => s.length())
res16:org.apache.spark.sql.UserDefinedFunction = UserDefinedFunction(<function1>,IntegerType)
//构建对列进行格式化的 UDF
scala > import java.text.DecimalFormat
import java.text.DecimalFormat
scala > sqlContext.udf.register("fmtNum",(d:Int) => {
     |                    valdfmt = new DecimalFormat("###.00")
     |                    dfmt.format(d.toDouble)})
res11:org.apache.spark.sql.UserDefinedFunction = UserDefinedFunction(<function1>,StringType)
```

注册的 strLen 的函数功能为将传入的字符串转换为字符串的长度，fmtNum 对 Int 型的 age 进行格式化。

在 SQL 中的使用案例如下：

```
scala > valudfRslt = sqlContext.sql("SELECT name,age FROM people WHERE strLen(name) <= 13
AND age <= 32").show
name      age
Michael 29
Andy      30
Justin   19
udfRslt:Unit = ()
```

在 DataFrame 中的 UDF 使用案例如下：

```
scala > people.selectExpr("strlen(name)","age").toDF("nameLen","age").show
nameLen age
7          29
4          30
6          19
scala > people.selectExpr("strlen(name) as nameLen","age").show
```



```
nameLen age
7       29
4       30
6       19
scala > people.selectExpr("strlen(name) as nameLen","age").printSchema
root
 | -- nameLen:integer (nullable = true)
 | --  age:integer (nullable = false)
scala > people.selectExpr("strlen(name) as nameLen","fmtNum(age)").show
nameLen fmtNum(age)
7       29.00
4       30.00
6       19.00
scala > people.selectExpr("strlen(name) as nameLen","fmtNum(age)").printSchema
root
 | -- nameLen:integer (nullable = true)
 | --' fmtNum(age):string (nullable = true)
```

这里使用注册的 strlen 这个 UDF 后，返回的 DataFrame 的 schema 也会自动进行推导。对应的 fmtNum 函数被调用后，age 字段也自动推导为 String 类型。

3.3.5 使用 JDBC 操作其他数据库的案例与解析

Spark SQL 还可以从其他数据库使用 JDBC 读取数据。这个功能应该优先于使用 jdbcRDD。因为它可以直接返回 DataFrame，方便在 Spark SQL 进行处理，也可以很容易地和其他数据源进行 join 操作。Java 或 Python 也很容易使用 JDBC 数据源，因为它不需要用户提供 ClassTag。注意，这和使用 Spark JDBC SQL 服务器是不同的，Spark JDBC SQL 服务器允许其他应用程序使用 Spark SQL 进行查询。

在案例操作开始前需要将对应数据库的 JDBC 驱动添加到 Spark 的 CLASSPATH 上。比如，在 Spark Shell 上连接 postgres 数据库时，需要使用下面的命令：

```
Spark_CLASSPATH = postgresql - 9.3 - 1102 - jdbc41.jar bin/spark - shell
```

为了解决提交应用时，经常出现的在 Executor 上找不到驱动包的问题，下面给出两种情况下的案例与分析。通过两种案例解析，可以熟悉不同场景下如何使用提交应用的参数选项。

一、集群中所有集群部署相同，并在集群上的某一节点启动 Driver Program

这种场景下，所有节点上都部署了 Spark、Hive，部署路径也相同，而且驱动类在 Hive 的 lib 目录下。这时候可以将 Hive 的 lib 目录下的驱动 jar 包添加到 Spark_CLASSPATH 中，添加时使用绝对路径，因此在每个节点上的 CLASSPATH 上都能在该绝对路径下找到驱动类。

当前案例以 MySQL 数据库为例，对应的命令为：

```
Spark_CLASSPATH =$Spark_CLASSPATH:/home/harli/cluster/hive/lib/mysql - connector - java -
5.1.35 - bin.jar ./bin/spark - shell
```

首先，登录 MySQL 客户端，查看当前 Hive 数据库下的表：

```
[harli@ cluster01 spark]$mysql  -uroot  -pmysql
Warning:Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.   Commands endwith ; or \g.
Your MySQL connection id is 12
Server version:5.6.23 MySQL Community Server (GPL)
Copyright (c) 2000,2015,Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type'help';or'\h' for help. Type'\c' to clear the current input statement.
mysql > use hive
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
mysql > show tables;
+---------------------------+
| Tables_in_hive            |
+---------------------------+
| BUCKETING_COLS            |
| CDS                       |
| COLUMNS_V2                |
| DATABASE_PARAMS           |
| DBS                       |
| FUNCS                     |
| FUNC_RU                   |
| GLOBAL_PRIVS              |
| PARTITIONS                |
| PARTITION_KEYS            |
| PARTITION_KEY_VALS        |
| PARTITION_PARAMS          |
| PART_COL_STATS            |
| ROLES                     |
| SDS                       |
| SD_PARAMS                 |
| SEQUENCE_TABLE            |
| SERDES                    |
| SERDE_PARAMS              |
| SKEWED_COL_NAMES          |
| SKEWED_COL_VALUE_LOC_MAP  |
| SKEWED_STRING_LIST        |
| SKEWED_STRING_LIST_VALUES |
| SKEWED_VALUES             |
| SORT_COLS                 |
| TABLE_PARAMS              |
```

```
|TAB_COL_STATS              |
|TBLS                       |
|VERSION                    |
+---------------------------+
29 rows in set (0.00 sec)
```

启动 spark - shell：

```
Spark_CLASSPATH =$Spark_CLASSPATH:/home/harli/cluster/hive/lib/mysql - connector - java -
5.1.35 - bin.jar ./bin/spark - shell  -- master spark://cluster01:7077
```

加载 JDBC 数据库 Hive 中的 TBLS 表。

```
scala > valjdbcDF = sqlContext.load("jdbc",Map(
     |     "url" ->"jdbc:mysql://192.168.242.131:3306/hive? user = root&password = mysql",
     |     "dbtable" ->"TBLS"))
jdbcDF:org.apache.spark.sql.DataFrame = [TBL_ID:bigint,CREATE_TIME:int,DB_ID:bigint,LAST_
ACCESS_TIME:int,OWNER:string,RETENTION:int,SD_ID:bigint,TBL_NAME:string,TBL_TYPE:
string,VIEW_EXPANDED_TEXT:string,VIEW_ORIGINAL_TEXT:string]
```

将 jdbcDF 注册为临时表：

```
scala > jdbcDF.registerTempTable("table")
scala > sqlContext.s
setConfsparkContextsql
scala > sqlContext.sql("select * from table")
15/04/06 06:15:31 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local
no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting
to a remote metastore.
15/04/06 06:15:31 INFO parse.ParseDriver:Parsing command:select * from table
15/04/06 06:15:32 INFO parse.ParseDriver:Parse Completed
res1:org.apache.spark.sql.DataFrame = [TBL_ID:bigint,CREATE_TIME:int,DB_ID:bigint,LAST_
ACCESS_TIME:int,OWNER:string,RETENTION:int,SD_ID:bigint,TBL_NAME:string,TBL_TYPE:
string,VIEW_EXPANDED_TEXT:string,VIEW_ORIGINAL_TEXT:string]
```

使用数据源 API，可以将远程数据库的表装载成一个 DataFrame 或 Spark SQL 的临时表。数据源 API 支持的选项如表 3.3 所示。

表 3.3　数据源 API 支持的选项

属　性　名	含　　义
url	要连接的 JDBC URL
dbtable	JDBC 的表，应该是可读的。注意，任何一个有效的 SQL 查询中的“FROM”子句都是可以使用的。比如，不使用整个表，而使用括号中的子查询语句
driver	连接到 URL 时需要 JDBC 驱动程序的类名。在运行一条 JDBC 命令让驱动器注册到 JDBC 子系统之前，master 和 workers 都需要先加载该类
partitionColumn，lowerBound，upperBound，numPartitions	如果需要指定这些选项，就必须同时全部指定。它们描述列在并行地从多个 worker 上读取数据时如何对表进行分区。其中，对表进行查询时的 partitionColumn 必须是一个数值型的列

二、在集群上的某一节点启动 Driver Program

这种场景下，需要保证两点，一是 Driver program 能找到驱动类，二是执行任务的 Executor 能找到驱动类。

使用第一种情况时，用 JDBC 表作为测试表。

1. 启动 spark - shell

```
[harli@ cluster01 spark]$./bin/spark - shell  - - master spark://cluster01:7077  - - driver - class - path ../hive/lib/mysql - connector - java - 5.1.35 - bin.jar  - - jars ../hive/lib/mysql - connector - java - 5.1.35 - bin.jar
```

命令参数说明：

1）通过 - - driver - class - path 选项，将 driver program 所在节点的驱动类路径加到 CLASSPATH 中；由于此时默认以 Client 部署模式提交，因此 Driver Program 在提交节点运行，所以这里可以使用相对路径。

2）通过 - - jars 选项，将 Executor 需要使用的 jar 包上传，由于上传的 jar 包会自动添加到执行点的 CLASSPATH，因此 Executor 执行时是可以识别的，不需要再手动添加到 CLASSPATH 上。这里也是把本地的驱动 jar 包作为 - - jars 参数。

注意：上传的 jar 包在执行时会自动下载。

2. 加载 JDBC 表

```
scala > val jdbcDF = sqlContext.load("jdbc",Map(
     |    "url" -> "jdbc:mysql://192.168.242.131:3306/hive? user = root&password = mysql",
     |    "dbtable" -> "TBLS"))
15/04/07 08:55:50 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/04/07 08:55:52 INFO metastore.HiveMetaStore:0:Opening raw store with implemenation class: org.apache.hadoop.hive.metastore.ObjectStore
15/04/07 08:55:52 INFO metastore.ObjectStore:ObjectStore,initialize called
15/04/07 08:55:52 WARN DataNucleus.General:Plugin (Bundle) "org.datanucleus.store.rdbms" is already registered. Ensure you dont have multiple JAR versions of the same plugin in the classpath. The URL "file:/home/harli/cluster/spark/lib/datanucleus - rdbms - 3.2.9.jar" is already registered,and you are trying to register an identical plugin located at URL "file:/home/harli/cluster_13/spark/lib/datanucleus - rdbms - 3.2.9.jar."
15/04/07 08:55:52 WARN DataNucleus.General:Plugin (Bundle) "org.datanucleus" is already registered. Ensure you dont have multiple JAR versions of the same plugin in the classpath. The URL "file:/home/harli/cluster_13/spark/lib/datanucleus - core - 3.2.10.jar" is already registered,and you are trying to register an identical plugin located at URL "file:/home/harli/cluster/spark/lib/datanucleus - core - 3.2.10.jar."
15/04/07 08:55:52 WARN DataNucleus.General:Plugin (Bundle) "org.datanucleus.api.jdo" is already registered. Ensure you dont have multiple JAR versions of the same plugin in the classpath. The URL "file:/home/harli/cluster/spark/lib/datanucleus - api - jdo - 3.2.6.jar" is already registered,and you are trying to register an identical plugin located at URL "file:/home/harli/cluster_13/spark/lib/datanucleus - api - jdo - 3.2.6.jar."
15/04/07 08:55:52 INFO DataNucleus.Persistence:Property datanucleus.cache.level2 unknown - will be ignored
```

```
15/04/07 08:55:52 INFO DataNucleus.Persistence:Property hive.metastore.integral.jdo.pushdown unknown - will be ignored
15/04/07 08:55:53 WARN DataNucleus.Connection:BoneCP specified but not present in CLASSPATH (or one of dependencies)
15/04/07 08:55:53 WARN DataNucleus.Connection:BoneCP specified but not present in CLASSPATH (or one of dependencies)
15/04/07 08:55:53 WARN conf.HiveConf:DEPRECATED:Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/04/07 08:55:53 INFO metastore.ObjectStore:Setting MetaStore object pin classes with hive.metastore.cache.pinobjtypes="Table,StorageDescriptor,SerDeInfo,Partition,Database,Type,FieldSchema,Order"
15/04/07 08:55:56 INFO DataNucleus.Datastore:The class "org.apache.hadoop.hive.metastore.model.MFieldSchema" is tagged as "embedded-only" so does not have its own datastore table.
15/04/07 08:55:56 INFO DataNucleus.Datastore:The class "org.apache.hadoop.hive.metastore.model.MOrder" is tagged as "embedded-only" so does not have its own datastore table.
15/04/07 08:55:56 INFO DataNucleus.Datastore:The class "org.apache.hadoop.hive.metastore.model.MFieldSchema" is tagged as "embedded-only" so does not have its own datastore table.
15/04/07 08:55:56 INFO DataNucleus.Datastore:The class "org.apache.hadoop.hive.metastore.model.MOrder" is tagged as "embedded-only" so does not have its own datastore table.
15/04/07 08:55:56 INFO DataNucleus.Query:Reading in results for query "org.datanucleus.store.rdbms.query.SQLQuery@0" since the connection used is closing
15/04/07 08:55:56 INFO metastore.ObjectStore:Initialized ObjectStore
15/04/07 08:55:57 INFO metastore.HiveMetaStore:Added admin role in metastore
15/04/07 08:55:57 INFO metastore.HiveMetaStore:Added public role in metastore
15/04/07 08:55:57 INFO metastore.HiveMetaStore:No user is added in admin role, since config is empty
15/04/07 08:55:58 INFO session.SessionState:No Tez session required at this point. hive.execution.engine=mr.
15/04/07 08:55:59 INFO session.SessionState:No Tez session required at this point. hive.execution.engine=mr.
jdbcDF:org.apache.spark.sql.DataFrame=[TBL_ID:bigint,CREATE_TIME:int,DB_ID:bigint,LAST_ACCESS_TIME:int,OWNER:string,RETENTION:int,SD_ID:bigint,TBL_NAME:string,TBL_TYPE:string,VIEW_EXPANDED_TEXT:string,VIEW_ORIGINAL_TEXT:string]
```

3. 显示 JDBC 表的内容。

```
scala>jdbcDF.show
15/04/07 08:56:06 INFO spark.SparkContext:Starting job:runJob at SparkPlan.scala:121
15/04/07 08:56:06 INFO scheduler.DAGScheduler:Got job 0 (runJob at SparkPlan.scala:121) with 1 output partitions (allowLocal=false)
15/04/07 08:56:06 INFO scheduler.DAGScheduler:Final stage:Stage 0(runJob at SparkPlan.scala:121)
15/04/07 08:56:06 INFO scheduler.DAGScheduler:Parents of final stage:List()
15/04/07 08:56:06 INFO scheduler.DAGScheduler:Missing parents:List()
15/04/07 08:56:06 INFO scheduler.DAGScheduler:Submitting Stage 0 (MapPartitionsRDD[1] at map at SparkPlan.scala:96),which has no missing parents
15/04/07 08:56:06 INFO storage.MemoryStore:ensureFreeSpace(4560) called with curMem=0,maxMem=280248975
15/04/07 08:56:06 INFO storage.MemoryStore:Block broadcast_0 stored as values in memory (esti-
```

```
mated size 4.5 KB,free 267.3 MB)
15/04/07 08:56:06 INFO storage.MemoryStore:ensureFreeSpace(2845) called with curMem=4560,
maxMem=280248975
15/04/07 08:56:06 INFO storage.MemoryStore:Block broadcast_0_piece0 stored as bytes in memory
(estimated size 2.8 KB,free 267.3 MB)
15/04/07 08:56:06 INFO storage.BlockManagerInfo:Added broadcast_0_piece0 in memory on clus-
ter01:49747 (size:2.8 KB,free:267.3 MB)
15/04/07 08:56:06 INFO storage.BlockManagerMaster:Updated info of block broadcast_0_piece0
15/04/07 08:56:06 INFO spark.SparkContext:Created broadcast 0 from broadcast at DAGScheduler.
scala:839
15/04/07 08:56:06 INFO scheduler.DAGScheduler:Submitting 1 missing tasks from Stage 0 (MapParti-
tionsRDD[1] at map at SparkPlan.scala:96)
15/04/07 08:56:06 INFO scheduler.TaskSchedulerImpl:Adding task set 0.0 with 1 tasks
15/04/07 08:56:07 INFO scheduler.TaskSetManager:Starting task 0.0 in stage 0.0 (TID 0,cluster01,
PROCESS_LOCAL,1140 bytes)
15/04/07 08:56:09 INFO storage.BlockManagerInfo:Added broadcast_0_piece0 in memory on clus-
ter01:49666 (size:2.8 KB,free:267.3 MB)
15/04/07 08:56:11 INFO scheduler.TaskSetManager:Finished task 0.0 in stage 0.0 (TID 0) in 4099
ms on cluster01 (1/1)
15/04/07 08:56:11 INFO scheduler.DAGScheduler:Stage 0 (runJob at SparkPlan.scala:121) finished
in 4.108 s
15/04/07 08:56:11 INFO scheduler.DAGScheduler:Job 0 finished:runJob at SparkPlan.scala:121,took
4.438742 s
15/04/07 08:56:11 INFO scheduler.TaskSchedulerImpl:Removed TaskSet 0.0,whose tasks have all
completed,from pool
TBL_ID CREATE_TIME DB_ID LAST_ACCESS_TIME OWNER RETENTION SD_ID TBL_NAME
TBL_TYPE        VIEW_EXPANDED_TEXT VIEW_ORIGINAL_TEXT
1       1428316407  1     0       harli 0       1     pokes       MANAGED_TABLE null
            null
6       1428320130  1     0       harli 0       6     pokes_test MANAGED_TABLE null
          null
```

注意：这里使用 spark-shell，只支持 Client 部署模式，如果使用 spark-submit 方式提交 Spark 应用程序的话，可以使用 Cluster 部署模式，这时候，Driver Program 会由 Master（Standalone 集群）或 ResourceManager（Spark on YARN）负责调度，此时，可以去掉针对本地 Driver Program 的 CLASSPATH 设置，即去掉 --driver-class-path 选项，--jars 上传的驱动类也会自动添加到实际运行节点上的 Driver Program 的 CLASSPATH 中。

另外，--jars 会随着应用上传，如果这种应用场景比较常用的话，建议使用配置属性，将驱动类的 jar 包部署到集群节点中。相关配置属性可以参考章节 2.1.2 应用程序的部署方式部分的提交脚本参数说明。

三、故障排除

当执行 Spark 应用程序时，如果出现 ClassNotFounded、Connect 建立失败，或者出现 SQL 语句解析异常等情况时，可以从以下两点进行故障排除：

1. JDBC 驱动程序类对在客户端会话和所有的 Executors 的初始类加载器必须是可见的。这是因为 Java 的 DriverManager 类进行安全检查，导致当建立一个连接时，会忽视所有对初始类加载器不可见的驱动类。一个方法是修改所有 worker 节点上的 compute_classpath.sh，

让其包含驱动 jar 包。

在包含驱动 jar 包时，需要注意 jar 包设置的路径应该为全路径。

比如，在 Spark_CLASSPATH 中添加 jar 包依赖时，必须使用实际执行时所在的路径；如果路径设置错误，会导致驱动查找失败而异常。异常信息可能包含以下内容：

```
15/04/06 23:54:57 INFO scheduler.DAGScheduler:Job 0 failed:runJob at SparkPlan.scala:121,took 2.265197 s
org.apache.spark.SparkException:Job aborted due to stage failure:Task 0 in stage 0.0 failed 4 times,
most recent failure:Lost task 0.3 in stage 0.0 (TID 3,cluster01):java.sql.SQLException:No suitable
driver found for jdbc:mysql://192.168.242.131:3306/hive? user=root&password=mysql
at java.sql.DriverManager.getConnection(DriverManager.java:596)
at java.sql.DriverManager.getConnection(DriverManager.java:233)
    …
```

2. 一些数据库，如 H2，会将所有名称转换为大写。在 Spark SQL 中，需要使用大写来引用那些名字。

3.3.6 集成 Hive 数据仓库的案例与解析

一般情况下，各个公司都会建立自己的数据仓库，尤其是当前大数据生态圈中使用最普遍的 Hive 数据仓库，需要集成这部分数据，向外提供这部分数据的查询接口。Spark SQL 提供了分布式 SQL 引擎，支持直接运行 SQL 查询的接口，不用写任何代码。

运行的集群环境说明：在新建的集群上运行，部署的 Spark 版本和 Hadoop 版本和之前的集群一样，并添加了 Hive 环境，Hive 版本为 apache-hive-1.1.0，对应数据库的驱动 jar 包为：mysql-connector-java-5.1.35.tar.gz。

一、Thrift JDBC/ODBC 的案例与解析

Spark SQL 提供 Thrift JDBC/ODBC 支持，这里实现的 Thrift JDBC/ODBC 服务器与 Hive 0.13 中的 HiveServer2 相一致。可以用在 Spark 或者 Hive 0.13 附带的 beeline 脚本测试 JDBC 服务器。

下面给出两种方式启动 JDBC/ODBC 服务的案例与解析。参考 Hive 的默认配置文件中的属性：

```
<property>
<name>hive.server2.transport.mode</name>
<value>binary</value>
<description>
    Expects one of [binary,http].
    Transport mode ofHiveServer2.
</description>
</property>
```

传输模式支持两种，Binary 和 HTTP，默认的为 Binary。

(一) 默认传输模式 (Binary)，启动 JDBC/ODBC 服务的案例与解析

这种情况下，在 hive-site.xml 中没有对 hive.server2.transport.mode 等属性进行修改。

在 Spark 目录中，运行下面的命令启动 JDBC/ODBC 服务器。

```
[harli@cluster01 spark]$./sbin/start-thriftserver.sh
starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /home/harli/cluster01/spark/sbin/../logs/spark-harli-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1-cluster01.out
```

这个脚本接受任何的 bin/spark-submit 命令行参数，加上一个 --hiveconf 参数用来指明 Hive 属性。可以运行“./sbin/start-thriftserver.sh --help”来获得所有可用选项的完整列表。默认情况下，服务器监听 localhost：10000。

可以用环境变量覆盖这些变量，如下所示。

```
//使用 export 设置环境变量
[harli@cluster01 spark]$export HIVE_SERVER2_THRIFT_PORT=10001
[harli@cluster01 spark]$export HIVE_SERVER2_THRIFT_BIND_HOST=cluster01
//设置环境变量后执行
[harli@cluster01 spark]$./sbin/start-thriftserver.sh \
> --master spark://cluster01:7077
starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /home/harli/cluster_13/spark/sbin/../logs/spark-harli-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1-cluster01.out
[harli@cluster01 spark]$./bin/beeline
Spark assembly has been built with Hive, includingDatanucleus jars on classpath
Beeline version 1.3.0 by Apache Hive
beeline>!connectjdbc:hive2://cluster01:10001
scan complete in 52ms
Connecting tojdbc:hive2://cluster01:10001
Enter username forjdbc:hive2://cluster01:10001:harli
Enter password forjdbc:hive2://cluster01:10001:
Connected to:Spark SQL (version 1.3.0)
Driver:Spark Project Core (version 1.3.0)
Transaction isolation:TRANSACTION_REPEATABLE_READ
0:jdbc:hive2://cluster01:10001>show tables;
+---------------------+--------------+
| tableName           | isTemporary  |
+---------------------+--------------+
| lpx_partition_test  | false        |
| partition_table     | false        |
| partition_test      | false        |
+---------------------+--------------+
3 rows selected (1.721 seconds)
0:jdbc:hive2://cluster01:10001>
```

设置 --master 参数后，应用提交到 Standaone 集群，查看界面可以看到应用信息，具体内容如图 3.21 所示。

当前使用默认参数，具体参数和 spark-submit 一样，可以根据需要调整。

也可以通过系统变量覆盖，命令如下：

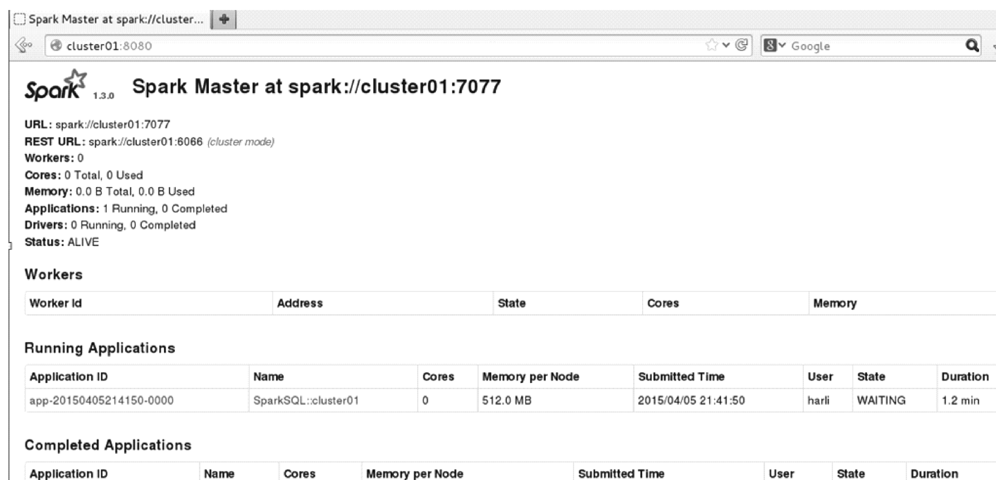


图 3.21 Spark 监控界面上的 Applications 信息

```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port=<listening-port> \
  --hiveconf hive.server2.thrift.bind.host=<listening-host> \
  --master <master-uri>
  ...
```

现在你可以用 beeline 来测试 Thrift JDBC/ODBC 服务器了。

在 Spark 部署主目录下输入：

```
[harli@cluster01 spark]$./bin/beeline
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
Beeline version 1.3.0 by Apache Hive
```

连接到 Thrift JDBC/ODBC 服务器的方式如下：

```
beeline>!connect jdbc:hive2://localhost:10000
scan complete in 19ms
Connecting tojdbc:hive2://localhost:10000
Enter username forjdbc:hive2://localhost:10000:harli
Enter password forjdbc:hive2://localhost:10000:*****
Connected to:Spark SQL (version 1.3.0)
Driver:Spark Project Core (version 1.3.0)
Transaction isolation:TRANSACTION_REPEATABLE_READ
```

这里的 localhost 和 10000 是默认的主机和端口号，可以用 Thrift JDBC/ODBC 服务启动时的实际主机和端口号进行替换。

Beeline 将会询问用户名和密码。在非安全的模式，简单地输入机器的用户名和空密码就可以。对于安全模式，可以按照 Beeline 文档的说明来执行。

注意：当报 URL 连接无效时，如果确认 URL 正确，可以查看进程或日志，先确认下 Thrift JDBC/ODBC 服务是否已经正常启动。

下面中 beeline 交互界面上执行基本的 SQL 语句，如下所示：

```
0:jdbc:hive2://localhost:10000 > show tables;
+---------------------+---------------+
| tableName           | isTemporary   |
+---------------------+---------------+
| lpx_partition_test  | false         |
| partition_table     | false         |
| partition_test      | false         |
+---------------------+---------------+
3 rows selected (2.098 seconds)
0:jdbc:hive2://localhost:10000 > select * from partition_table;
+-------+
| age   |
+-------+
+-------+
No rows selected (1.59 seconds)
```

show tables 可以看到之前用 SQLContext 在 Hive 中建立的所有表信息。

最后向 beeline 发送 quit 命令，退出：

```
0:jdbc:hive2://cluster01:10001 > !quit
Closing:0:jdbc:hive2://cluster01:10001
```

（二）HTTP 传输模式启动 JDBC/ODBC 服务的案例与解析

Thrift JDBC 服务业支持通过 HTTP 传输模式来发送 thrift RPC 消息。可以通过将 HTTP 模式的配置设置成系统属性，或修改 conf 下的 hive-site.xml 文件中的属性配置来实现，如下所示。

```
<property>
<name>hive.server2.transport.mode</name>
<value>http</value>
<description>
        Expects one of [binary,http].
        Transport mode ofHiveServer2.
</description>
</property>
<property>
<name>hive.server2.thrift.http.port</name>
<value>10001</value>
<description>Port number ofHiveServer2 Thrift interface when hive.server2.transport.mode is' http' .
</description>
</property>
<property>
<name>hive.server2.thrift.http.path</name>
<value>cliservice</value>
<description>Path component of URL endpoint when in HTTP mode.</description>
</property>
```

其中，最后一个为 hive.server2.thrift.http.path 属性（官网上的属性在默认 hive-default.xml 中没有找到）。添加后，复制到“$Spark_HOME/conf”目录下。



编辑 hive2server 启动脚本：

```
nohup ./bin/hiveserver2 >> hiveserver2.log 2 > &1 &
echo$! > hive-server2.pid
```

启动 Thrift Server 测试：

```
[harli@ cluster01 spark]$  ./sbin/start-thriftserver.sh --master spark://cluster01:7077
starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /home/harli/cluster_13/spark/sbin/../logs/spark-harli-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1-cluster01.out
```

启动 beeline 连接、查询：

```
[harli@ cluster01 spark]$./bin/beeline
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
Beeline version 1.3.0 by Apache Hive
beeline >! connectjdbc:hive2://cluster01:10001/default? hive.server2.transport.mode = http;hive.server2.thrift.http.path = cliservice
scan complete in 1ms
Connecting tojdbc:hive2://cluster01:10001/default? hive.server2.transport.mode = http;hive.server2.thrift.http.path = cliservice
Enter username forjdbc:hive2://cluster01:10001/default? hive.server2.transport.mode = http;hive.server2.thrift.http.path = cliservice:root
Enter password forjdbc:hive2://cluster01:10001/default? hive.server2.transport.mode = http;hive.server2.thrift.http.path = cliservice:*****
Connected to:Spark SQL (version 1.3.0)
Driver:Spark Project Core (version 1.3.0)
Transaction isolation:TRANSACTION_REPEATABLE_READ
0:jdbc:hive2://cluster01:10001/default > show tables;
+-------------+--------------+
| tableName   | isTemporary  |
+-------------+--------------+
| pokes       | false        |
| pokes_test  | false        |
| t           | false        |
+-------------+--------------+
3 rows selected (0.194 seconds)
0:jdbc:hive2://cluster01:10001/default > select * from t;
+----+----+----+
| a  | b  | c  |
+----+----+----+
+----+----+----+
No rows selected (1.184 seconds)
```

这里也可以试试启动 Hive 的服务 Hive2server，如下所示：

```
[harli@ cluster01 hive]$./bin/start-hiveserver2.sh
[harli@ cluster01 hive]$jps
5023 Worker
```

```
4575 SecondaryNameNode
9059 Jps
4349 DataNode
4793 Master
8520 BeeLine
4168 NameNode
8971 RunJar
7445 RunJar
```

可以看到，这里增加了一个 RunJar 的进程。由于 hivemetastore 的进程名字一样，都是 RunJar，Hive 脚本中没有直接提供停止服务的操作，因此这里在脚本中将 pid 信息写入了各自的文件中。通过查看该文件，获取 pid 值，然后关闭进程。

启动 HTTP 模式下的 beeline：

```
[harli@ cluster01 spark]$./bin/beeline
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
Beeline version 1.3.0 by Apache Hive
//连接
beeline >! connectjdbc: hive2 ://cluster01 : 10001/default? hive.server2.transport.mode = http;
hive.server2.thrift.http.path = cliservice
scan complete in 1ms
Connecting to jdbc: hive2 ://cluster01 : 10001/default? hive.server2.transport.mode = http;
hive.server2.thrift.http.path = cliservice
Enter username for jdbc: hive2 ://cluster01 : 10001/default? hive.server2.transport.mode = http;
hive.server2.thrift.http.path = cliservice:root
Enter password for jdbc: hive2 ://cluster01 : 10001/default? hive.server2.transport.mode = http;
hive.server2.thrift.http.path = cliservice: *****
Connected to:Apache Hive (version 1.1.0)
Driver:Spark Project Core (version 1.3.0)
Transaction isolation:TRANSACTION_REPEATABLE_READ
//查询默认数据库中的表
0:jdbc:hive2://cluster01:10001/default > show tables;
+--------------+
|  tab_name    |
+--------------+
| pokes        |
| pokes_test   |
| t            |
+--------------+
3 rows selected (1.168 seconds)
//查询 pokes 表的数据
0:jdbc:hive2://cluster01:10001/default > select * from pokes;
+------------+------------+
| pokes.foo  | pokes.bar  |
+------------+------------+
+------------+------------+
No rows selected (1.161 seconds)
```

这里的 10001 为刚才设置属性 hive.server2.thrift.http.port，default 为默认的数据库。连

接成功，并查询出当前 Hive 中已经有三个表。

二、Spark SQL CLI 的案例与解析

Spark SQL CLI 是一个便利的工具，它可以在本地运行 Hive 元存储服务、执行命令行输入的查询。注意，Spark SQL CLI 不能与 Thrift JDBC 服务器通信。

在 Spark 目录运行下面的命令可以启动 Spark SQL CLI：

```
./bin/spark - sql
```

注意：如果 Hive 使用内置的默认 derby 数据库，同时只能允许一个会话连接，则启动 Thrift JDBC 服务后再次启动 Spark - sql 将会报异常，异常信息会包含以下内容：

```
……
Caused by: java.sql.SQLException: Failed to start database' metastore _ db' with class loader sun.misc.Launcher $AppClassLoader@46aea8cf, see the next exception for details.
    at org.apache.derby.impl.jdbc.SQLExceptionFactory.getSQLException(Unknown Source)
      at org.apache.derby.impl.jdbc.SQLExceptionFactory40.wrapArgsForTransportAcrossDRDA ( Unknown Source)
...77 more
Caused by: java.sql.SQLException: Another instance of Derby may have already booted the database /home/harli/cluster_13/spark/metastore_db.
……
```

连接 Hive 时，在 CLASSPATH 中添加数据库驱动 jar 包，输入命令：

```
Spark_CLASSPATH = $Spark_CLASSPATH:/home/harli/cluster/hive/lib/mysql - connector - java - 5.1.35 - bin.jar./bin/spark - sql
```

查询当前表信息：

```
spark - sql > show tables;
15/04/06 06:19:29 INFOmetastore.HiveMetaStore:0:get_tables:db = default pat =. *
15/04/06 06:19:29 INFOHiveMetaStore.audit:ugi = harliip = unknown - ip - addrcmd = get_tables:db
   = default pat =. *
15/04/06 06:19:29 INFO spark.SparkContext:Starting job:collect at SparkPlan.scala:83
……
15/04/06 06:19:29 INFO scheduler.StatsReportListener:0%   5%   10%   25%   50%   75%
   90%   95%   100%
15/04/06 06:19:29 INFO scheduler.StatsReportListener:95%   95%   95%   95%   95%   95%
   95%   95%   95%
15/04/06 06:19:29 INFO scheduler.DAGScheduler:Job 2 finished:collect at
SparkPlan.scala:83,took 0.110593 s
pokesfalse
pokes_testfalse
Time taken:0.13 seconds,Fetched 2 row(s)
15/04/06 06:19:29 INFOCliDriver:Time taken:0.13 seconds,Fetched 2 row(s)
```

查询成功，就可以开始执行其他的 SQL 语句了。

Section 3.4 基于 Hive 的人力资源系统数据处理案例与解析

这一节给出一个简单的公司人力资源系统的数据处理案例与解析。通过该案例，给出一

个比较完整的、复杂的数据处理案例，同时给出案例的详细解析。包括以下几个部分：

1）人力资源系统的数据库与表的构建。

2）人力资源系统的数据的加载。

3）人力资源习题的数据的查询。

当前人力资源管理系统的管理内容组织结构如图 3.22 所示。

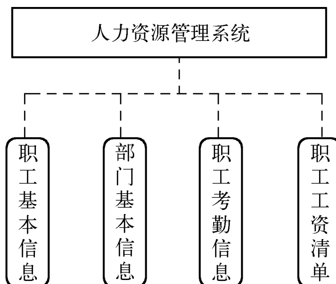


图 3.22　人力资源管理系统内容组织结构图

当前人力资源系统的数据包含以下几个部分：

1. 职工基本信息：存放职工的基本信息，包含职工姓名，职工 id，职工性别，职工年龄，入职年份，职位，所在部门 id 等信息；数据内容如下：

```
Michael,1,male,37,2001,developer,2
Andy,2,female,33,2003,manager,1
Justin,3,female,23,2013,recruitingspecialist,3
John,4,male,22,2014,developer,2
Herry,5,male,27,2010,developer,1
Brewster,6,male,37,2001,manager,2
Brice,7,female,30,2003,manager,3
Justin,8,male,23,2013,recruitingspecialist,3
John,9,male,22,2014,developer,1
Herry,10,female,27,2010,recruitingspecialist,3
```

2. 部门基本信息：存放部门信息，包含部门名称，编号，数据内容如下：

```
management,1
researchanddevelopment,2
HumanResources,3
```

3. 职工考勤信息：存放职工的考勤信息，包含年、月信息，职工加班，迟到，旷工，早退小时数信息。

```
1,2015,12,0,2,4,0
2,2015,8,5,0,5,3
3,2015,3,16,4,1,5
4,2015,3,0,0,0,0
5,2015,3,0,3,0,0
6,2015,3,32,0,0,0
7,2015,3,0,16,3,32
8,2015,19,36,0,0,0,3
9,2015,5,6,30,0,2,2
10,2015,10,6,56,40,0,22
1,2014,12,0,2,4,0
2,2014,38,5,40,5,3
3,2014,23,16,24,1,5
4,2014,23,0,20,0,0
5,2014,3,0,3,20,0
6,2014,23,32,0,0,0
7,2014,43,0,16,3,32
```

```
8,2014,49,36,0,20,0,3
9,2014,45,6,30,0,22,2
10,2014,40,6,56,40,0,22
```

4. 职工工资清单：存放职工每月的工资清单信息。



```
1,5000
2,10000
3,6000
4,7000
5,5000
6,11000
7,12000
8,5500
9,6500
10,4500
```

3.4.1 人力资源系统的数据库与表的构建

将人力资源系统的数据加载到 Hive 仓库的 HRS 数据库中，并对人力资源系统的数据分别建表。

1. 启动 spark - shell

```
bin/spark - shell -- executor - memory 2g -- driver - memory 1g -- master
spark://cluster01:7077
```

其中，cluster01 为当前 Spark 的 Master 节点。

由于当前使用 Hive 作为数据仓库，因此需要参考章节 3.3.6 集成 Hive 数据仓库的案例与解析准备环境，即已经进行 hive - site. xml 文件配置并启动了 metastore 服务等准备操作。

除去多余的日志信息：

```
scala > import org. apache. log4j. {Level,Logger}
import org. apache. log4j. {Level,Logger}
scala > Logger. getLogger("org. apache. spark"). setLevel(Level. WARN)
scala > Logger. getLogger("org. apache. spark. sql"). setLevel(Level. WARN)
scala > Logger. getLogger("org. apache. hadoop. hive"). setLevel(Level. WARN)
```

以应用程序方式提交时，可以在配置文件 conf/log4j. properties 中设置日志等级，如下所示：

```
log4j. logger. org. apache. spark = WARN
log4j. logger. org. apache. spark. sql = WARN
log4j. logger. org. apache. hadoop. hive. ql = WARN
```

2. 构建与使用 HRS 数据库

1）使用 CREATE DATABASE 语句创建名为 HRS 的数据库，存放人力资源系统的数据：

```
scala > sqlContext. sql("CREATE DATABASE HRS")
15/05/18 09:49:51 WARNHiveConf:DEPRECATED:Configuration property
```

```
hive. metastore. local no longer has any effect. Make sure to provide a valid value for
hive. metastore. uris if you are connecting to a remote metastore.
15/05/18 09:49:51 INFOParseDriver:Parsing command:CREATE DATABASE HRS
15/05/18 09:49:51 INFOParseDriver:Parse Completed
15/05/18 09:49:51 INFOParseDriver:Parsing command:CREATE DATABASE HRS
15/05/18 09:49:51 INFOParseDriver:Parse Completed
res10:org. apache. spark. sql. DataFrame = [result:string]
```

2）使用人力资源系统的数据库 HRS：

```
scala > sqlContext. sql("USE HRS")
15/05/18 09:50:33 WARNHiveConf:DEPRECATED:Configuration property
hive. metastore. local no longer has any effect. Make sure to provide a valid value for
hive. metastore. uris if you are connecting to a remote metastore.
15/05/18 09:50:33 INFOParseDriver:Parsing command:USE HRS
15/05/18 09:50:33 INFOParseDriver:Parse Completed
15/05/18 09:50:33 INFOParseDriver:Parsing command:USE HRS
15/05/18 09:50:33 INFOParseDriver:Parse Completed
res11:org. apache. spark. sql. DataFrame = [result:string]
```

3. 数据建表

创建四个数据对应的表，仅在表不存在时创建（表存在时不创建）。

1）构建职工基础信息表 people：

```
scala > sqlContext. sql("CREATE TABLE IF NOT EXISTS people(name STRING,id
INT,gender STRING,age INT,year INT,position STRING,depID INT) ROW FORMAT
DELIMITED FIELDS TERMINATED BY' ', LINES TERMINATED BY' \n' ")
15/05/19 10:52:58 WARNHiveConf:DEPRECATED:Configuration property
hive. metastore. local no longer has any effect. Make sure to provide a valid value for
hive. metastore. uris if you are connecting to a remote metastore.
15/05/19 10:52:58 INFOParseDriver:Parsing command:CREATE TABLE IF NOT
EXISTS people(name STRING,id INT,gender STRING,age INT,year INT,position
STRING,depID INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY' ', LINES
TERMINATED BY'
'
15/05/19 10:52:58 INFOParseDriver:Parse Completed
15/05/19 10:52:58 INFOParseDriver:Parsing command:CREATE TABLE IF NOT
EXISTS people(name STRING,id INT,gender STRING,age INT,year INT,position
STRING,depID INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY' ', LINES
TERMINATED BY'
'
15/05/19 10:52:58 INFOParseDriver:Parse Completed
15/05/19 10:52:58 INFODDLTask:Default to LazySimpleSerDe for table people
res10:org. apache. spark. sql. DataFrame = [result:string]
```

2）构建部门基础信息表 department：

```
scala > sqlContext. sql("CREATE TABLE IF NOT EXISTS department(name STRING,depID INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY' ', LINES TERMINATED BY' \n' ")
```

```
res11:org. apache. spark. sql. DataFrame = [result:string]
```

3）构建职工考勤信息表 attendance：



```
scala > sqlContext. sql( "CREATE TABLE IF NOT EXISTS attendance (id INT,year INT,
month INT,overtime INT,latetime INT,absenteeism INT,leaveearlytime INT) ROW
FORMAT DELIMITED FIELDS TERMINATED BY' ', LINES TERMINATED BY' \n' ")

res12:org. apache. spark. sql. DataFrame = [result:string]
```

4）构建职工工资清单表 salary：

```
scala > sqlContext. sql( "CREATE TABLE IF NOT EXISTS salary (id INT,salary INT) ROW
FORMAT DELIMITED FIELDS TERMINATED BY' ', LINES TERMINATED BY' \n' ")
res13:org. apache. spark. sql. DataFrame = [result:string]
```

3.4.2 人力资源系统的数据的加载

分别将本地文本文件的数据加载到四个表。

职工基础信息表的加载操作如下：

```
scala > sqlContext. sql( "LOAD DATA LOCAL INPATH
' /home/harli/cluster/data/people. txt' OVERWRITE INTO TABLE people ")
15/05/19 10:55:09 WARNHiveConf:DEPRECATED:Configuration property
hive. metastore. local no longer has any effect. Make sure to provide a valid value for
hive. metastore. uris if you are connecting to a remote metastore.
15/05/19 10:55:09 INFOParseDriver:Parsing command:LOAD DATA LOCAL INPATH
' /home/harli/cluster/data/people. txt' OVERWRITE INTO TABLE people
15/05/19 10:55:09 INFOParseDriver:Parse Completed
15/05/19 10:55:09 INFOParseDriver:Parsing command:LOAD DATA LOCAL INPATH
' /home/harli/cluster/data/people. txt' OVERWRITE INTO TABLE people
15/05/19 10:55:09 INFOParseDriver:Parse Completed
rmr:DEPRECATED:Please use' rm - r' instead.
15/05/19 10:55:12 INFOTrashPolicyDefault:Namenode trash configuration:Deletion
interval =0 minutes,Emptier interval =0 minutes.
Deletedhdfs://cluster01:9000/user/hive/warehouse/hrs. db/people
15/05/19 10:55:12 INFO Hive:Replacing
src:hdfs://cluster01:9000/tmp/hive - harli/hive_2015 - 05 - 19_10 - 55 - 09_848_554477476
7724946117 - 1/ - ext - 10000;dest:
hdfs://cluster01:9000/user/hive/warehouse/hrs. db/people;Status:true
res14:org. apache. spark. sql. DataFrame = [result:string]
```

其中 OVERWRITE 表示覆盖当前表的数据，即先清除表数据，再将数据 insert 到表中。其他表的加载操作类似，具体如下：

```
//加载部门基础信息表数据
sqlContext. sql( "LOAD DATA LOCAL INPATH' /home/harli/cluster/data/department. txt'
INTO TABLE department ")
//加载职工考勤信息表数据
```

```
sqlContext.sql("LOAD DATA LOCAL INPATH' /home/harli/cluster/data/attendance.txt'
INTO TABLE attendance")
//加载职工工资清单表数据
sqlContext.sql("LOAD DATA LOCAL INPATH' /home/harli/cluster/data/salary.txt' INTO
TABLE salary")
```

3.4.3 人力资源系统的数据的查询

人力资源系统的数据常见的查询操作有部门职工数的查询、部门职工的薪资 topN 的查询、部门职工平均工资的排名、各部门每年职工薪资的总数查询等，下面给出具体的操作方法。

查看各表的信息，同时查看界面回显中的 schema 信息：

```
scala > sqlContext.sql("select * from people")
res5:org.apache.spark.sql.DataFrame = [name:string,id:int,gender:string,age:int,
year:int,position:string,depid:int]
scala > sqlContext.sql("select * from department")
res6:org.apache.spark.sql.DataFrame = [name:string,depid:int]
scala > val attendance = sqlContext.sql("select * from attendance")
attendance:org.apache.spark.sql.DataFrame = [id:int,year:int,month:int,overtime:
int,latetime:int,absenteeism:int,leaveearlytime:int]
scala > val salary = sqlContext.sql("select * from salary")
salary:org.apache.spark.sql.DataFrame = [id:int,salary:int]
```

1. 部门职工数的查询

首先将 people 表数据与 department 表数据进行 join 操作，然后根据 department 的部门名进行分组，分组后针对 people 中唯一标识一个职工的 id 字段进行统计，最后得到各个部门对应的职工总数统计信息。

```
scala > sqlContext.sql("select b.name,count(a.id) from people a join department b on a.depid =
b.depid group by b.name").show
15/05/19 10:56:01 WARNHiveConf:DEPRECATED:Configuration property hive.metastore.local no
longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a
remote metastore.
15/05/19 10:56:01 INFOParseDriver:Parsing command:select b.name,count(a.id) from people a join
department b on a.depid = b.depid group by b.name
15/05/19 10:56:01 INFOParseDriver:Parse Completed
15/05/19 10:56:02 INFO deprecation:mapred.map.tasks is deprecated. Instead,use mapreduce.job.
maps
15/05/19 10:56:02 WARNHiveConf:DEPRECATED:Configuration property hive.metastore.local no
longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a
remote metastore.
15/05/19 10:56:02 INFOFileInputFormat:Total input paths to process:1
15/05/19 10:56:08 WARNHiveConf:DEPRECATED:Configuration property hive.metastore.local no
longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a
remote metastore.
15/05/19 10:56:08 INFOFileInputFormat:Total input paths to process:1
```

```
name                    _c1
HumanResources          4
researchanddevelo…      3
management              3
```



2. 对各个部门职工薪资的总数、平均值的排序

首先根据部门 id 将 people 表数据与 department 表数据进行 join 操作，根据职工 idjoinsalary 表数据，然后根据 department 的部门名进行分组，分组后针对职工的薪资进行求和或求平均值，并根据该值大小进行排序（默认排序为从小到大）。

```
scala > sqlContext. sql( " select b. name, sum( c. salary) as s from people a join department b on a. depid = b. depid join salary c on a. id = c. id group by b. name order by s" ). show
15/05/19 11: 01: 16 WARNHiveConf: DEPRECATED: Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/05/19 11:01:16 INFOParseDriver: Parsing command: select b. name, sum( c. salary) as s from people a join department b on a. depid = b. depid join salary c on a. id = c. id group by b. name order by s
15/05/19 11:01:16 INFOParseDriver: Parse Completed
15/05/19 11:01:16 INFOFileInputFormat: Total input paths to process: 1
15/05/19 11: 01: 17 WARNHiveConf: DEPRECATED: Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/05/19 11:01:17 INFOFileInputFormat: Total input paths to process: 1
15/05/19 11:01:17 INFOFileInputFormat: Total input paths to process: 1
name                    s
management              21500
researchanddevelo…23000
HumanResources          28000
```

查询各个部门职工薪资的平均值的排序如下：

```
scala > sqlContext. sql( " select b. name, avg( c. salary) as s from people a join department b on a. depid = b. depid join salary c on a. id = c. id group by b. name order by s" ). show
15/05/20 11:43:07 WARNHiveConf: DEPRECATED: Configuration property
hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/05/20 11:43:07 INFOParseDriver: Parsing command: select b. name, avg( c. salary) as s from people a join department b on a. depid = b. depid join salary c on a. id = c. id group by b. name order by s
15/05/20 11:43:07 INFOParseDriver: Parse Completed
15/05/20 11: 43: 08 INFO deprecation: mapred. map. tasks is deprecated. Instead, use mapreduce. job. maps
15/05/20 11: 43: 08 WARNHiveConf: DEPRECATED: Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/05/20 11: 43: 09 WARNHiveConf: DEPRECATED: Configuration property hive. metastore. local no longer has any effect. Make sure to provide a valid value for hive. metastore. uris if you are connecting to a remote metastore.
15/05/20 11:43:09 INFOFileInputFormat: Total input paths to process: 1
15/05/20 11:43:09 INFOFileInputFormat: Total input paths to process: 1
```

```
15/05/20 11:43:16 WARNHiveConf:DEPRECATED:Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/05/20 11:43:16 INFOFileInputFormat:Total input paths to process:1
name                  s
HumanResources        7000.0
management            7166.666666666667
researchanddevelo...7666.666666666667
```

3. 查询各个部门职工的考勤信息

首先根据职工 id 将 attendance 考勤表数据与 people 职工表数据进行 join 操作，并计算职工的考勤信息，然后根据 department 的部门名、考勤信息的年份进行分组，分组后针对职工的考勤信息进行统计。

```
scala > sqlContext.sql("select b.name,sum(h.attdinfo),h.year from (select a.id,a.depid,at.year,at.month,overtime - latetime - absenteeism - leaveearlytime as attdinfo from attendance at join people a on at.id = a.id) h join department b on h.depid = b.depid group by b.name,h.year").show
15/05/22 10:08:40 WARNHiveConf:DEPRECATED:Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/05/22 10:08:40 INFOParseDriver:Parsing command:select b.name,sum(h.attdinfo),h.year from (select a.id,a.depid,at.year,at.month,overtime - latetime - absenteeism - leaveearlytime as attdinfo from attendance at join people a on at.id = a.id) h join department b on h.depid = b.depid group by b.name,h.year
15/05/22 10:08:40 INFOParseDriver:Parse Completed
15/05/22 10:08:40 INFOFileInputFormat:Total input paths to process:1
15/05/22 10:08:40 INFOFileInputFormat:Total input paths to process:1
15/05/22 10:08:40 INFOFileInputFormat:Total input paths to process:1
name                  _c1   year
management            -112  2014
management            -32   2015
HumanResources        -139  2014
HumanResources        -99   2015
researchanddevelo...6       2014
researchanddevelo...26      2015
```

其中，返回结果中的第一行表示字段名，_c1 为新增的考勤信息统计结果字段名；其他行表示对应字段的值。

4. 合并前面的全部查询

```
scala > sqlContext.sql("select e.name,e.pcount,f.sumsalary,f.avgsalary,j.year,j.sumattd from (select b.name,count(a.id) as pcount from people a join department b on a.depid = b.depid group by b.name order by pcount) e join (select b.name,sum(c.salary) as sumsalary,avg(c.salary) as avgsalary from people a join department b on a.depid = b.depid join salary c on a.id = c.id group by b.name order by sumsalary) f on (e.name = f.name) join (select b.name,sum(h.attdinfo) as sumattd,h.year from (select a.id,a.depid,at.year,at.month,overtime - latetime - absenteeism - leaveearlytime as attdinfo from attendance at join people a on at.id = a.id) h join department b on h.depid = b.depid group by b.name,h.year) j on f.name = j.name order by f.name").show
15/05/22 10:17:31 WARNHiveConf:DEPRECATED:Configuration property hive.metastore.local no
```

```
longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/05/22 10:17:31 INFOParseDriver: Parsing command: select e.name, e.pcount, f.sumsalary, f.avgsalary, j.year, j.sumattd from (select b.name, count(a.id) as pcount from people a join department b on a.depid = b.depid group by b.name order by pcount) e join (select b.name, sum(c.salary) as sumsalary, avg(c.salary) as avgsalary from people a join department b on a.depid = b.depid join salary c on a.id = c.id group by b.name order by sumsalary) f on (e.name = f.name) join (select b.name, sum(h.attdinfo) as sumattd, h.year from (select a.id, a.depid, at.year, at.month, overtime - latetime - absenteeism - leaveearlytime as attdinfo from attendance at join people a on at.id = a.id) h join department b on h.depid = b.depid group by b.name, h.year) j on f.name = j.name order by f.name
15/05/22 10:17:31 INFOParseDriver: Parse Completed
15/05/22 10:17:31 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:31 WARNHiveConf: DEPRECATED: Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/05/22 10:17:31 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:31 WARNHiveConf: DEPRECATED: Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/05/22 10:17:31 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:31 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:31 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:32 WARNHiveConf: DEPRECATED: Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.
15/05/22 10:17:32 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:32 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:32 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:32 INFOFileInputFormat: Total input paths to process: 1
15/05/22 10:17:34 INFOFileInputFormat: Total input paths to process: 1
name                    pcountsumsalaryavgsalary          year sumattd
HumanResources          4       28000     7000.0               2014 -139
HumanResources          4       28000     7000.0               2015 -99
management              3       21500     7166.666666666667    2014 -112
management              3       21500     7166.666666666667    2015 -32
researchanddevelo...3     23000     7666.666666666667 2014 6
researchanddevelo...3     23000     7666.666666666667 2015 26
```

将前面的几个查询合并到一个 SQL 语句中，最后得到部门的各种统计信息，包括部门职工数、部门薪资、部门每年的考勤统计等信息。

第4章 Spark Streaming实践案例与解析

4.1 Spark Streaming 概述
4.2 Spark Streaming 基础概念
4.3 企业信息实时处理的案例与解析
4.4 性能调优



Section 4.1 Spark Streaming 概述

Spark Streaming 是一种构建在 Spark 上的实时计算框架，它扩展了 Spark 处理大规模流式数据的能力。对应在伯克利数据分析协议栈中的位置如图 4.1 所示。

Spark Streaming 是 Spark 核心 API 的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括 Kafka、Flume、Twitter、ZeroMQ、Kinesis 以及 TCP sockets，从数据源获取数据之后，可以使用诸如 map、reduce、join 和 window 等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统，数据库和现场仪表盘上。在“One Stack rule them all”的基础上，还可以使用 Spark 的其他子框架，如集群学习、图计算等，对流数据进行处理。

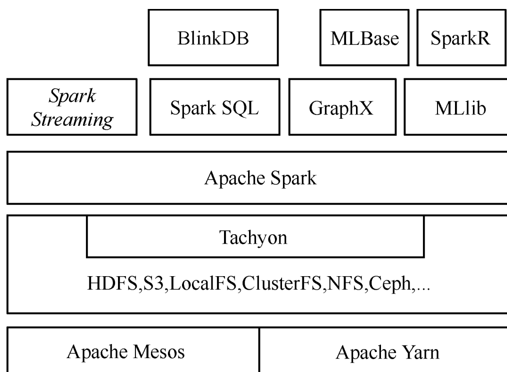


图 4.1　伯克利数据分析协议栈

官网提供的 Spark Streaming 处理的数据流如图 4.2 所示。

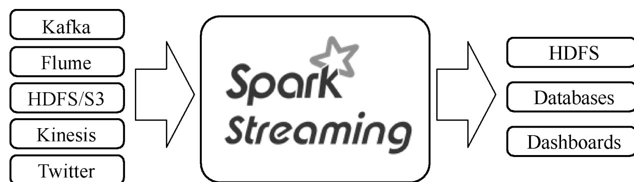


图 4.2　Spark Streaming 的数据流图

Spark 的各个子框架，都是基于核心 Spark 的，Spark Streaming 在内部的处理机制是，接收实时流的数据，并根据一定的时间间隔拆分成一批批的数据，然后通过 Spark Engine 处理这些批数据，最终得到处理后的一批批结果数据。

对应的批数据，在 Spark 内核对应一个 RDD 实例，因此，对应流数据的 DStream 可以看成是一组 RDD，即 RDD 的一个序列。通俗点理解的话，在流数据分成一批一批后，通过一个先进先出的队列，然后 Spark Engine 从该队列中依次取出一个个批数据，把批数据封装成一个 RDD，然后进行处理，这是一个典型的生产者 – 消费者模型，对应的就有生产者 – 消费者模型的问题，即如何协调生产速率和消费速率。

官网上给出 Spark Streaming 的内部处理机制流程如图 4.3 所示。

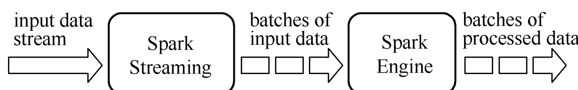


图 4.3　Spark Streaming 的内部处理机制流程图

对 DStream 的操作会对应到 DStream 底层的 RDD 序列的操作，内部的处理机制可以用图 4.4 描述。

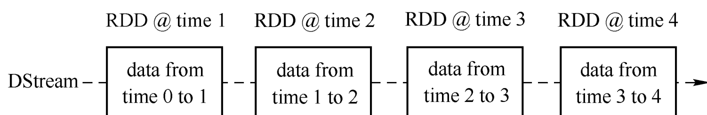


图 4.4 DStream 内部的处理机制

作用在 DStream 上的 API，在内部，实际上是作用到这个 RDD 序列上。

Section 4.2 Spark Streaming 基础概念

为了更好地理解 Spark Streaming 子框架的处理机制，先简单描述下相关的基础概念。

1. 离散流（Discretized Stream，DStream）：这是 Spark Streaming 对内部持续的实时数据流的抽象描述，即我们处理的一个实时数据流，在 Spark Streaming 中对应于一个 DStream 实例。

2. 批数据（batch data）：这是化整为零的第一步，将实时流数据以时间片为单位进行分批，将流处理转化为时间片数据的批处理。随着持续时间的推移，这些处理结果就形成了对应的结果数据流了。

3. 时间片或批处理时间间隔（batch interval）：这是人为地对流数据进行定量的标准，以时间片作为拆分流数据的依据。一个时间片的数据对应一个 RDD 实例。

4. 窗口长度（window length）：一个窗口覆盖的流数据的时间长度。必须是批处理时间间隔的倍数。

5. 滑动时间间隔：前一个窗口到后一个窗口所经过的时间长度。必须是批处理时间间隔的倍数。

6. input DStream：一个 input DStream 是一个特殊的 DStream，将 Spark Streaming 连接到一个外部数据源来读取数据。

7. Receiver：长时间（可能 7×24 小时）运行在 Executor。每个 Receiver 负责一个 input DStream（例如一个读取 Kafka 消息的输入流）。每个 Receiver，加上 input DStream 会占用一个 core/slot。

Section 4.3 企业信息实时处理的案例与解析

作为一个企业级的实时流处理应用，如果缺乏与诸如 Kafka 或 Flume 等进行整合的话，这种大数据流处理应用可以说是不完整的。

本节以 Spark Streaming 的外部数据源，Kafka 和 Flume 进行案例实践与分析，建立接入外部数据源的基础案例之后，可以在数据的处理上使用 Spark Streaming 或 RDD 提供的高层次 API 对数据进行业务相关的处理。

构建 Spark Streaming 应用程序和开发一个 Spark 的应用程序一样，需要依赖于 Spark

Streaming 的 jar 包，下面给出构建 Spark Streaming 应用程序的三种方式。

1. 基于 SBT 进行构建

需要在对应的构建文件中添加依赖：

```
libraryDependencies += "org. apache. spark" % "spark - streaming_2. 10" % "1. 3. 0"
```

2. 基于 Maven 进行构建

需要在对应的构建文件中添加依赖：

```
<dependency>
<groupId>org. apache. spark</groupId>
<artifactId>spark - streaming_2. 10</artifactId>
<version>1. 3. 0</version>
</dependency>
```

3. 使用 IDEA 进行构建

通过 IDEA 添加 Libraries，这部分的详细操作可以参考章节 2. 4. 2 基于 IDEA 构建 Spark 应用程序的案例部分。在后面的 TCP 数据源案例与解析部分，也会给出添加 Spark 部署的全部 lib 下的 jar 包的案例。

说明：由于 SBT 和 Maven 的构建方式类似，后续将基于 SBT 和 IDEA 给出基础的应用程序构建案例与分析。

4. 3. 1 处理 TCP 数据源的案例与解析

一、准备工程，并构建测试类

（一）基于 IDEA 构建应用程序

在第 2 章构建的工程基础上，参考章节 2. 4. 2 基于 IDEA 构建 Spark 应用程序的实例部分，继续添加依赖包，如图 4. 5 所示。

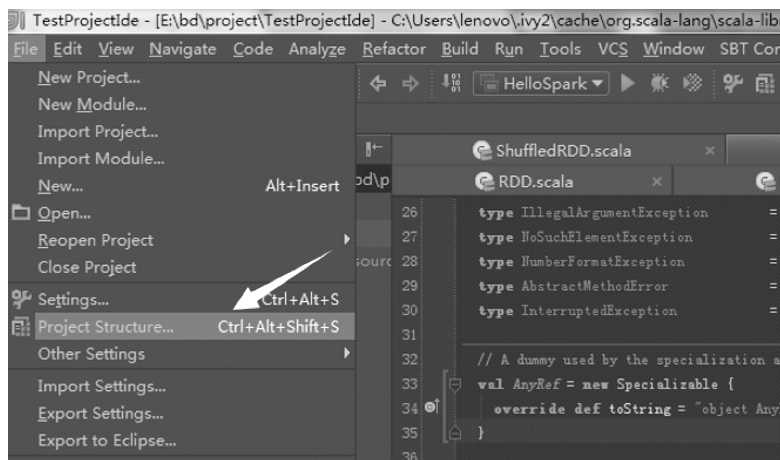


图 4. 5　IDEA 中的 Project Structure…选项

在 IDEA 中添加依赖包，如图 4. 6 所示。

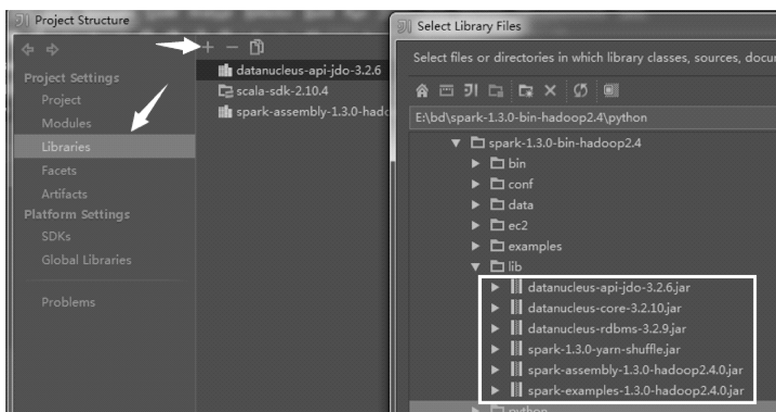


图 4.6　IDEA 中添加的依赖包

这里我们基于 examples 提供的 NetworkWordCount 类来实践 TCP 流数据的处理，相应的，为 Spark - examples - 1.3.0 - hadoop2.4.0.jar 添加源码关联，查找 examples 中的 Network-WordCount 类，查找结果如图 4.7 所示。

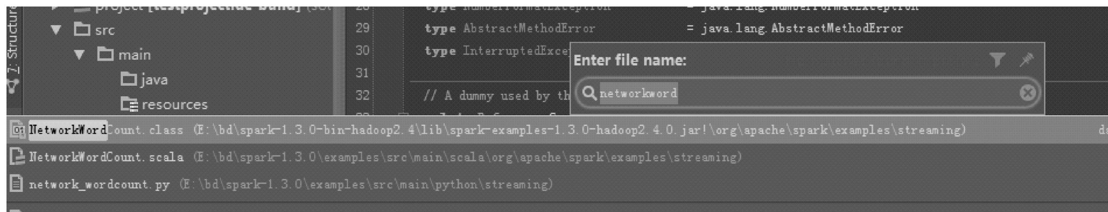


图 4.7　IDEA 中查找 NetworkWordCount 类

构建自己的 package，名为 stream，在 scala 目录下，单击右键打开上下文菜单，依次选择 New→Package 命令，操作步骤如图 4.8 所示。

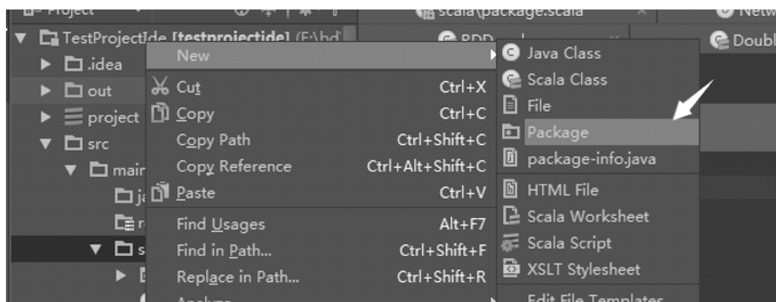


图 4.8　IDEA 中添加 package

输入 stream 作为 package 名，单击 OK 按钮，如图 4.9 所示。

构建 package 后的目录结构如图 4.10 所示。

目录结构中，在 stream 上单击右键，在弹出窗口中创建一个名为 NetworkWordCount 的对象，如图 4.11 所示。

点击 OK 按钮，复制代码，如图 4.12 所示。

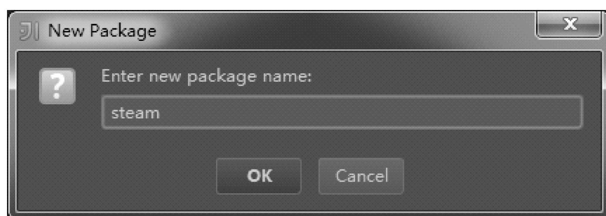


图 4.9　IDEA 中设置添加的 package 名称

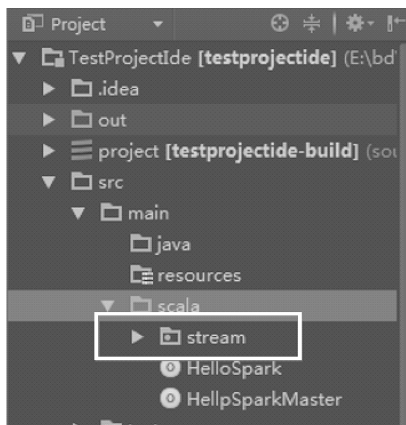


图 4.10　IDEA 中构建 package 后的目录结构

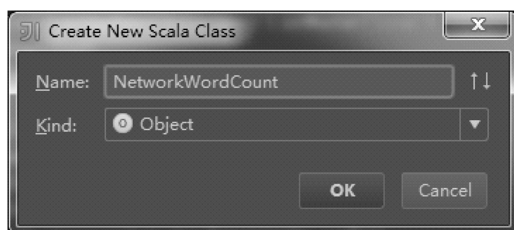


图 4.11　IDEA 中创建一个 NetworkWordCount 对象

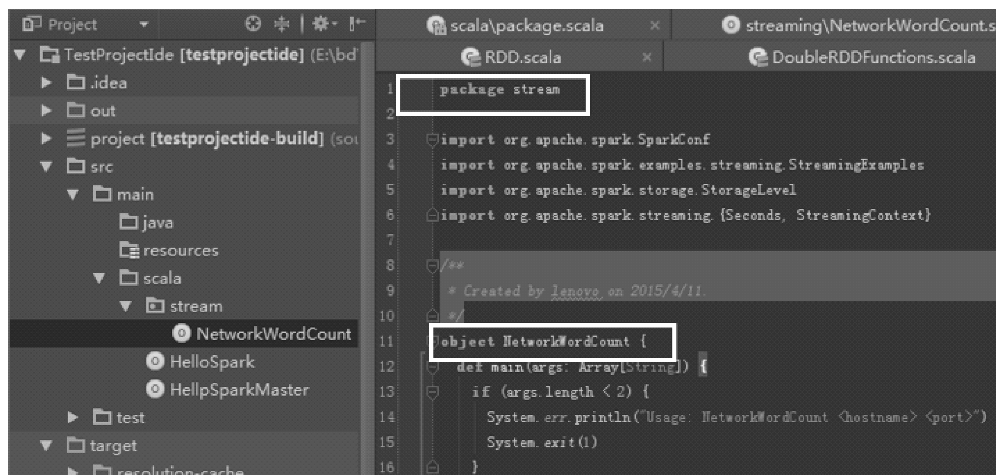


图 4.12　IDEA 中复制 NetworkWordCount 对象的代码

构建应用程序的 jar 包，如图 4.13 所示。这里的 Artifacts 参见章节 2.4.2 基于 IDEA 构建 Spark 应用程序的实例部分。

查看构建的 jar 包，可以看到已经包含了 NetworkWordCount 类了，包含内容如图 4.14 所示。

可以通过 WinRAR 等解压工具打开 jar 包进行查看，也可以在命令行中使用 jar 命令来解压查看，使用方法和 tar 类似，具体可以查看命令的帮助信息。

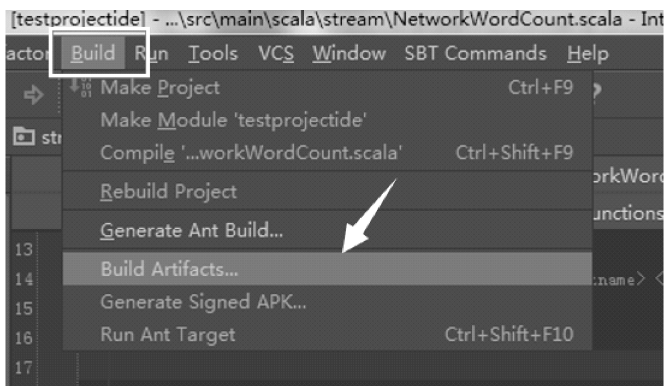


图 4.13　IDEA 中构建应用程序的 jar 包

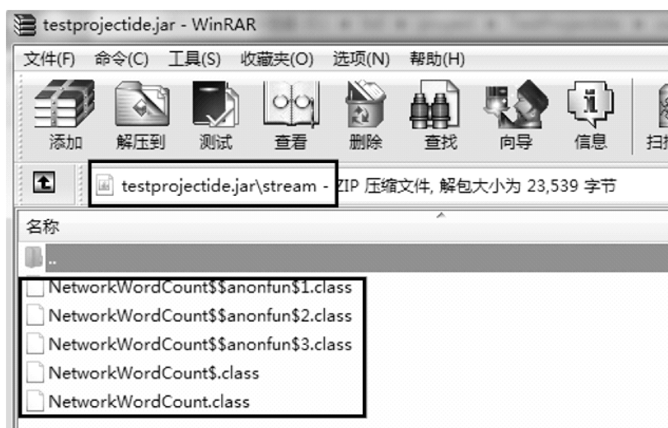


图 4.14　查看构建的 jar 包的类

（二）基于 SBT 构建应用程序

在 build. sbt 文件中添加：

```
libraryDependencies += "org. apache. Spark" %% "Spark – streaming" % "1. 3. 0"
```

注意：这里添加依赖的语法和前面有点差异，使用的“%%”，同时不需要指定 Scala 版本，这在比较新的 SBT 版本中才支持，具体可以参考 SBT 官方网站。

打开终端，如 IDEA 中的终端 Terminal（也可以打开 Windows 下的 CMD 窗口），输入命令“sbt package”，具体操作如图 4.15 所示。

编译成功的输出信息如图 4.16 所示。

由于当前使用的 Maven 仓库中没有对应的 Spark – examples 的 1.3 版本的 jar 包，如图 4.17 所示。

在 SBT 编译前修改代码，注释掉//StreamingExamples. setStreamingLogLevels( )，或将自己编译得到的 jar 包放入本地仓库中（可以在编译时加 install），或者直接将 Spark 部署包中的 Spark – examples – 1. 3. 0 – hadoop2. 4. 0. jar 复制到本地仓库中。SBT 打包应用程序可以参考章节 2. 4. 1 基于 SBT 构建 Spark 应用程序的实例部分。

这里使用 IDEA 方式构建应用程序，手动将依赖的 jar 包导入 IDEA 中，然后构建出应用

```
Terminal
E:\bd\project\TestProjectIde> sbt package
[info] Loading global plugins from C:\Users\lenovo\.sbt\0.13\plugins
[info] Loading project definition from E:\bd\project\TestProjectIde\project
[info] Set current project to TestProjectIde (in build file:/E:/bd/project/TestPro
[info] Updating {file:/E:/bd/project/TestProjectIde/}testprojectide...
[info] Resolving org.scala-lang#scala-library;2.10.4 ...
  [info] Resolving org.apache.spark#spark-core_2.10;1.3.0 ...
  [info] Resolving com.twitter#chill_2.10;0.5.0 ...
  [info] Resolving com.twitter#chill-java;0.5.0 ...
  [info] Resolving com.esotericsoftware.kryo#kryo;2.21 ...
  [info] Resolving com.esotericsoftware.reflectasm#reflectasm;1.07 ...
6: TODO   SBT Console   Terminal
```

图 4.15　sbt package 方式构建 jar 包

```
Terminal
  [info] Resolving org.apache.spark#spark-streaming_2.10;1.3.0 ...
  [info] Resolving commons-net#commons-net;3.1 ...
  [info] Resolving org.slf4j#slf4j-api;1.7.5 ...
  [info] Resolving org.slf4j#slf4j-api;1.7.10 ...
  [info] Resolving org.scala-lang#scala-compiler;2.10.4 ...
  [info] Resolving org.scala-lang#jline;2.10.4 ...
  [info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[success] Total time: 23 s, completed 2015-4-12 0:15:57

E:\bd\project\TestProjectIde>
```

图 4.16　sbt package 方式构建 jar 包的结果界面

repo1.maven.org/maven2/org/apache/spark/spark-examples_2.10/

应用　编程语言　数据处理　搜索　Tools　项目、版本　harli　博客　others　网站　Work　Bigdata　github/ap

Index of /maven2/org/apache/spark/spark-examples_2.10/

../		
0.9.0-incubating/	31-Jan-2014 06:05	-
0.9.1/	08-Apr-2014 02:50	-
0.9.2/	23-Jul-2014 02:54	-
1.0.0/	29-May-2014 22:43	-
1.0.1/	11-Jul-2014 16:56	-
1.0.2/	04-Aug-2014 23:11	-
1.1.0/	08-Sep-2014 01:04	-
1.1.1/	01-Dec-2014 20:36	-
maven-metadata.xml	01-Dec-2014 20:36	570
maven-metadata.xml.md5	01-Dec-2014 20:36	32
maven-metadata.xml.sha1	01-Dec-2014 20:36	40

图 4.17　maven 仓库中 jar 包信息

程序的 jar 包，StreamingExamples.setStreamingLogLevels()这一行代码在实际测试过程中并不能去除日志信息，所以暂时使用以下代码来替代（可通过日志的配置文件进行修改，但不推荐在生产环境中去除 INFO 级别的日志）：

```
import org.apache.log4j.{Level,Logger}
Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
```

```
Logger.getLogger("org.apache.spark.streaming").setLevel(Level.WARN)
```

最终的应用程序 NetworkWordCount 的代码如下：

```
objectNetworkWordCount {
  def main(args:Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage:NetworkWordCount <hostname> <port>")
      System.exit(1)
    }
StreamingExamples.setStreamingLogLevels()
import org.apache.log4j.{Level,Logger}
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.streaming").setLevel(Level.WARN)
    //设置批数据的时间片大小为 1s
    valsparkConf = new SparkConf().setAppName("NetworkWordCount")
    val ssc = new StreamingContext(sparkConf,Seconds(1))
//使用输入的 host 和 port 构建 Socket 流,并设置存储级别
    //构建后得到 DStream 实例
    val lines = ssc.socketTextStream(args(0),args(1).toInt,StorageLevel.MEMORY_AND_DISK_SER)
    //DStream 提供和 RDD 类似的 high - level 的 api 对内部 RDD 序列进行处理
    //下面的处理方式和 RDD 的单词统计是一样的
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x,1)).reduceByKey(_ + _)
wordCounts.print()
    //最后需要调用 start 来正式启动流处理
    ssc.start()
ssc.awaitTermination()
  }
}
```

任何作用在 DStream 实例上的操作都会转换为对其底层 RDD 序列的操作，比如，代码中 flatMap 方法对应的 DStream 内部操作如图 4.18 所示。

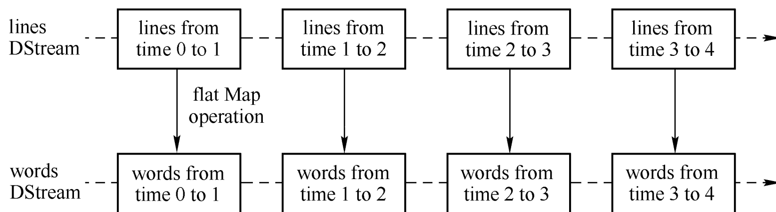


图 4.18　DStream 的 flatMap 方法对应的内部操作

其中，一个框对应一个批数据，即一个 RDD 实例。

二、开始测试

这里使用 cluster01 的集群，cluster01 对应部署了全部进程，除了集群设备较少之外，其他操作和多节点集群是一样的。

（一）Standalone 模式提交

启动 Spark 集群后，在$Spark_HOME 路径下输入：

```
[harli@cluster01 spark]$./bin/spark-submit  --exector-memory 1g\
        --master spark://cluster01:7077\
        --class stream.NetworkWordCount\
        --jars./lib/spark-examples-1.3.0-hadoop2.4.0.jar../applitions/testprojectide.jar cluster01
9999
```

根据 NetworkWordCount 应用的使用说明"Usage:NetworkWordCount <hostname> <port>"，在 spark-submit 的最后输入对应的 cluster01 9999，作为应用程序的参数。

需要注意的是，由于 NetworkWordCount 代码中使用了 StreamingExamples 类，因此需要将依赖的./lib/spark-examples-1.3.0-hadoop2.4.0.jar 作--jars 参数传入，否则 Executor 执行时会找不到 StreamingExamples 类，错误信息如下：

```
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
Exception in thread "main" java.lang.NoClassDefFoundError:org/apache/spark/examples/streaming/
StreamingExamples$
        at stream.DirectKafkaWordCount$.main(KafkaDirectApp.scala:58)
        at stream.DirectKafkaWordCount.main(KafkaDirectApp.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:601)
        at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$run-
Main(SparkSubmit.scala:569)
        at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:166)
        at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:189)
        at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:110)
        at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
Caused by: java.lang.ClassNotFoundException:
org.apache.spark.examples.streaming.StreamingExamples$
        at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:423)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
```

注意：这里使用 spark-examples-1.3.0-hadoop2.4.0.jar 主要是为了测试在不同集群模式下，用--jars 添加依赖包的有效性（集群中没有将 Spark 的 lib 放到 Hadoop 的 CLASSPATH 路径下）。

查看界面，如图 4.19 所示。由于当前虚拟机设置了 2 个内核，因此该应用占用的内核数为 2，刚好可以分配给 Executor 和接收流的 Receiver。

为了方便测试，接下来将 cluster01 的虚拟机内核数改成 4。

打开另一个终端，输入以下命令启动 Netcat：

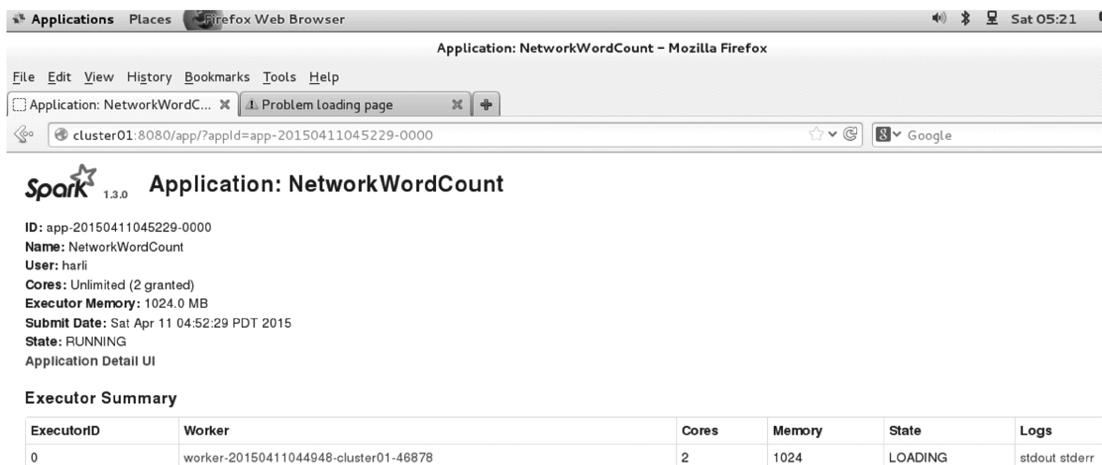


图 4.19　Spark 应用程序界面信息

```
[harli@ cluster01Spark]$nc -lk 9999
```

然后将$Spark_HOME 路径下的 README. md 内容复制到该终端界面上。

切换到 Spark -submit 的终端，可以看到如下输出：

```
-------------------------------------------
Time:1428754186000 ms
-------------------------------------------
(package,1)
(this,1)
(Because,1)
(Python,2)
(cluster. ,1)
(its,1)
([run,1)
(YARN,,1)
(general,2)
(have,1)
…
-------------------------------------------
Time:1428754187000 ms
-------------------------------------------

-------------------------------------------
Time:1428754188000 ms
-------------------------------------------

-------------------------------------------
Time:1428754189000 ms
-------------------------------------------
(configure,1)
(how,1)
(,2)
```



```
(documentation,1)
(on,1)
(to,1)
(Spark. ,1)
(in,1)
(online,1)
(for,1)
…
```

可以在 Time 处看到每隔 1s 提交一次 job 进行单词统计，这里还有统计时没有收到数据但也提交 job 的，后续会在案例中给出处理空 RDD 的方法。

（二）Yarn 模式提交

首先停止之前的 Spark - submit 命令，用【Ctrl + C】组合键停止，然后使用 jps 命令查询：

```
[harli@ cluster01Spark]$jps
13645DataNode
23702SparkSubmit
22925 CoarseGrainedExecutorBackend
21744 Worker
13812SecondaryNameNode
17705RunJar
23832 Jps
21524 Master
13513NameNode
```

使用 kill 命令终止 SparkSubmit 进程：

```
[harli@ cluster01Spark]$kill -9 23702
```

这里使用 Client 部署模式提交，可以不用【Ctrl + C】组合键，而是直接在另一个终端上查询 pid 然后使用 kill 命令关闭：

```
ps - aux |grep SparkSubmit
kill -9 $pid      //pid 为 ps 后的进程 ID 值
```

启动 Yarn 服务：

```
[harli@ cluster01 hadoop]$. /sbin/start - yarn. sh
starting yarn daemons
startingresourcemanager,logging to /home/harli/cluster_13/hadoop/logs/yarn - harli - resourcemanager
 - cluster01. out
cluster01:startingnodemanager,logging to /home/harli/cluster_13/hadoop/logs/yarn - harli - nodeman-
ager - cluster01. out
```

到$Spark_HOME 路径下，再次提交命令：

```
. /bin/spark - submit   - - executor - memory 1g - - master yarn - - deploy - mode client - - class
stream. NetworkWordCount - - jars. /lib/spark - examples - 1. 3. 0 - hadoop2. 4. 0. jar. . /applitions/
testprojectide. jar cluster01 9999
```

注意：由于 Hadoop 上也没有部署 spark-examples-1.3.0-hadoop2.4.0.jar，因此需要使 --jars 参数进行上传。

在 nc 终端继续将 README.md 文件内容复制进去，再次看到 NetworkWordCount 应用输出单词统计信息：

```
(detailed,1)
(GraphX,1)
(package,1)
(this,1)
(stream,1)
(is,3)
(its,1)
(general,2)
(pre-built,1)
(built,1)
...
```

打开 Hadoop 的 ResourceManager 监控界面，查看应用提交结果，如图 4.20 所示。

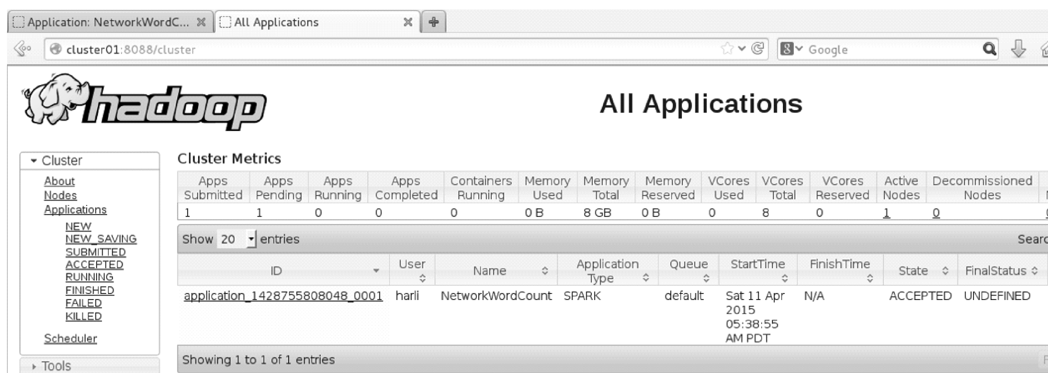


图 4.20　HadoopResourceManager 监控界面的应用程序信息

ResourceManager 监控界面地址为：http://cluster01:8088，其中 cluster01 是启动 ResourceManager 进程的节点。

切换成 yarn-cluster 方式提交时，查看 Hadoop 界面，如图 4.21 所示。

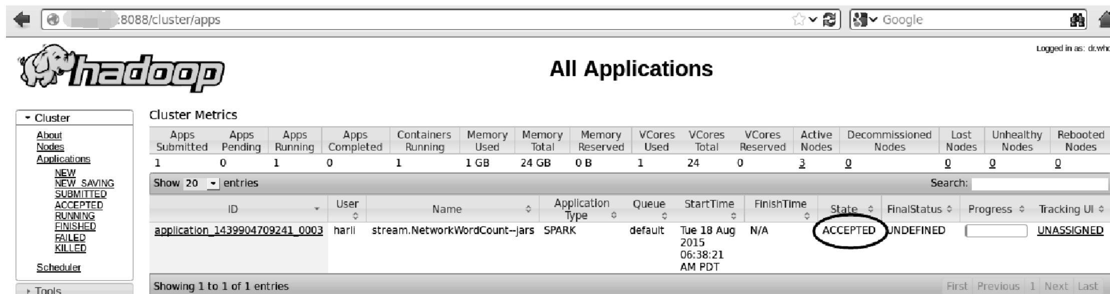


图 4.21　HadoopResourceManager 监控界面的 AM 信息

提交成功，单击进入应用后，如图 4.22 所示出现界面。

Spark Jobs (?)

Total Duration: 1.5 min
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 155

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total
5	start at NetworkWordCount.scala:37	2015/04/11 06:06:49	1.2 min	0/1

图 4.22　Spark 的 job 信息

继续查看 Executors 信息，如图 4.23 所示。

Executors (3)

Memory: 0.0 B Used (1306.3 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
1	cluster01:35285	0	0.0 B / 530.3 MB	0.0 B	1	0	30	31	2.3 s	0.0 B	1881.0 B	3.0 KB	stdout stderr	Thread Dump
2	cluster01:53707	0	0.0 B / 530.3 MB	0.0 B	0	0	674	674	17.2 s	0.0 B	1539.0 B	5.3 KB	stdout stderr	Thread Dump
<driver>	cluster01:39836	0	0.0 B / 245.7 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

图 4.23　Spark 的 Executors 信息

当前虚拟机的内核数为 4，Yarn 模式下，提交时 Executor 的个数默认为 2，分配内核为 1，因此总的使用内核数为 2，对应加一个 ApplicationMaster，即图中的 driver，一共对应 3 个内核。

在图 4.23 中可看到在 Hadoop 界面对应的 driver 中没有 Logs 信息 stdout 和 stderr，不过可以在终端打开该应用下的日志。查看 driver 的日志输出信息，和我们用 Client 方式提交时的界面信息是一样的。

当前运行的 driver 程序日志所在路径为：

/home/harli/cluster/hadoop/**logs**/userlogs/**application_1428755808048_0004**/container_1428755808048_0004_01_000001

logs 目录位于执行 driver 的节点上（在 Addresse 列可以看到当前执行的节点），其中，application_1428755808048_0004 对应应用程序的 ID。

对应在 Yarn 模式下执行的应用程序，可以用以下命令关闭，相关的几个进程也会被关闭：

```
hadoop job -kill application_1428755808048_0004
```

上面是旧的版本所使用的命令，也可以用较新版本的命令，如下所示：

```
yarn application -kill application_1428755808048_0004
```

（三）Yarn 模式提交补充案例

由于前面 yarn-cluster 提交的案例是在单机上模拟集群进行的，这时候依赖的第三方包在集群中都是相同位置。同时，针对第三方包的依赖具体过程的疑问，如同时提交出现类找不到等问题，这里为了进一步详细描述，用最新的集群（在基于 Tachyon 实践案例与解析基础上部署的新的集群——3 个节点上），重新给出案例，并在案例中给出详细的包的上传、下载的路径信息。

首先，启动 Yarn 服务：

```
[harli@cluster04 hadoop]$./sbin/start-yarn.sh
starting yarn daemons
startingresourcemanager,logging to /home/harli/cluster_13/hadoop/logs/yarn-harli-resourcemanager
-cluster04.out
cluster06:startingnodemanager,logging to /home/harli/cluster_13/hadoop/logs/yarn-harli-nodeman-
ager-cluster06.out
cluster05:startingnodemanager,logging to /home/harli/cluster_13/hadoop/logs/yarn-harli-nodeman-
ager-cluster05.out
cluster04:startingnodemanager,logging to /home/harli/cluster_13/hadoop/logs/yarn-harli-nodeman-
ager-cluster04.out
```

启动命令：

```
./bin/Spark-submit    --executor-memory 1g \
 --master yarn-cluster \
 --num-executors 3 \
 --class stream.NetworkWordCount \
 --jars./Spark-examples-1.3.0-hadoop2.6.0.jar../applications/testprojectide.jar cluster04 9999
```

这里以 yarn-cluster 模式提交，同时，启动了 3 个 Executor。
在另一个终端启动 nc，并输入要统计的数据：

```
[harli@cluster04Spark]$nc -lk 9999
15/05/09 11:39:16 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
15/05/09 11:39:17 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
15/05/09 11:39:18 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
15/05/09 11:39:19 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
15/05/09 11:39:20 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
15/05/09 11:39:21 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
15/05/09 11:39:22 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
15/05/09 11:39:23 INFO Client:Application report for application_1431196702641_0001 (state:RUN-
NING)
```

```
15/05/09 11:39:24 INFO Client:Application report for application_1431196702641_0001 (state:RUNNING)
15/05/09 11:39:25 INFO Client:Application report for application_1431196702641_0001 (state:RUNNING)
15/05/09 11:39:26 INFO Client:Application report for application_1431196702641_0001 (state:RUNNING)
15/05/09 11:39:27 INFO Client:Application report for application_1431196702641_0001 (state:RUNNING)
```

启动 TCP 流处理命令：

```
[harli@cluster04 spark]$ ./bin/spark-submit --executor-memory 1g --master yarn-cluster --num-executors 3 --class stream.NetworkWordCount --jars ./spark-examples-1.3.0-hadoop2.6.0.jar ../applications/testprojectide.jar cluster04 9999
Spark assembly has been built with Hive, including Datanucleus jars on classpath
15/05/09 11:38:53 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/05/09 11:38:53 INFO RMProxy: Connecting to ResourceManager at cluster04/192.168.242.135:8032
15/05/09 11:38:53 INFO Client: Requesting a new application from cluster with 3 NodeManagers
15/05/09 11:38:53 INFO Client: Verifying our application has not requested more than the maximum memory capability of the cluster (8192 MB per container)
15/05/09 11:38:53 INFO Client: Will allocate AM container, with 896 MB memory including 384 MB overhead
15/05/09 11:38:53 INFO Client: Setting up container launch context for our AM
15/05/09 11:38:53 INFO Client: Preparing resources for our AM container
15/05/09 11:38:55 INFO Client: Uploading resource file:/home/harli/cluster_13/spark/lib/spark-assembly-1.3.0-hadoop2.6.0.jar -> hdfs://cluster04:9000/user/harli/.sparkStaging/application_1431196702641_0001/spark-assembly-1.3.0-hadoop2.6.0.jar
15/05/09 11:38:58 INFO Client: Uploading resource file:/home/harli/cluster_13/applications/testprojectide.jar -> hdfs://cluster04:9000/user/harli/.sparkStaging/application_1431196702641_0001/testprojectide.jar
15/05/09 11:38:58 INFO Client: Uploading resource file:/home/harli/cluster_13/spark/spark-examples-1.3.0-hadoop2.6.0.jar -> hdfs://cluster04:9000/user/harli/.sparkStaging/application_1431196702641_0001/spark-examples-1.3.0-hadoop2.6.0.jar
15/05/09 11:39:00 INFO Client: Setting up the launch environment for our AM container
```

可以看到，在 yarn-cluster 模式提交时，会将依赖的 jar 包和主资源 jar 包一起上传到 HDFS 上。

查看上传后的路径下的文件：

```
[harli@cluster04 spark]$hdfs dfs -ls hdfs://cluster04:9000/user/harli/.sparkStaging/application_1431196702641_0001/
Found 3 items
-rw-r--r--   1 harli supergroup  167828287 2015-05-09 11:38 hdfs://cluster04:9000/user/harli/.sparkStaging/application_1431196702641_0001/spark-assembly-1.3.0-hadoop2.6.0.jar
-rw-r--r--   1 harli supergroup  114040423 2015-05-09 11:39 hdfs://cluster04:9000/user/harli/.sparkStaging/application_1431196702641_0001/spark-examples-1.3.0-hadoop2.6.0.jar
-rw-r--r--   1 harli supergroup      96168 2015-05-09 11:38 hdfs://cluster04:9000/user/harli/.sparkStaging/application_1431196702641_0001/testprojectide.jar
```

可以看到文件已经成功上传。

查看各个执行节点上的缓存文件，这里以 cluster06 节点为例，其包含文件如下：

```
[harli@ cluster06 Desktop]$cd
/home/harli/cluster/hadoop/tmp/nm - local - dir/usercache/harli/filecache
[harli@ cluster06 filecache]$ll - R. /
. /:
total 12
drwxr - xr - x. 2 harliharli 4096 May   9 11:39 10
drwxr - xr - x. 2 harliharli 4096 May   9 11:39 11
drwxr - xr - x. 2 harliharli 4096 May   9 11:39 12

. /10:
total 111368
 - r - x ------. 1harliharli 114040423 May   9 11:39 spark - examples - 1. 3. 0 - hadoop2. 6. 0. jar

. /11:
total 163896
 - r - x ------. 1harliharli 167828287 May   9 11:39 spark - assembly - 1. 3. 0 - hadoop2. 6. 0. jar

. /12:
total 96
 - r - x ------. 1harliharli 96168 May   9 11:39 testprojectide. jar
```

可以看到，执行节点已经成功将所依赖的 jar 包下载到 NodeManager 的本地路径下，为应用提供依赖 jar 包。

其中，nm - local - dir 是 NodeManager 执行应用时的 local 目录，执行时应该从 HDFS 上下载下来，并存放到该目录下。

这里给出不同于前面的另一种查看输出日志的方法。

1）进入 RM 节点的 Web Interface 界面（http://cluster04:8088/cluster），如图 4.24 所示。

图 4.24　HadoopRM 的应用信息

2）单击 application_1431196702641_0001，查看 Application 的具体信息，如图 4.25 所示。

3）单击 cluster04:8042，查看 Node 节点具体信息，如图 4.26 所示。

4）单击 List of Containers，查看容器信息，如图 4.27 所示。

图 4.25 HadoopRM 的指定应用的信息

图 4.26 HadoopRM 的指定 node 的信息

图 4.27 HadoopRM 的指定应用的容器的信息

5）单击 container_1431196702641_0001_01_000001，查看容器具体信息，如图 4.28 所示。

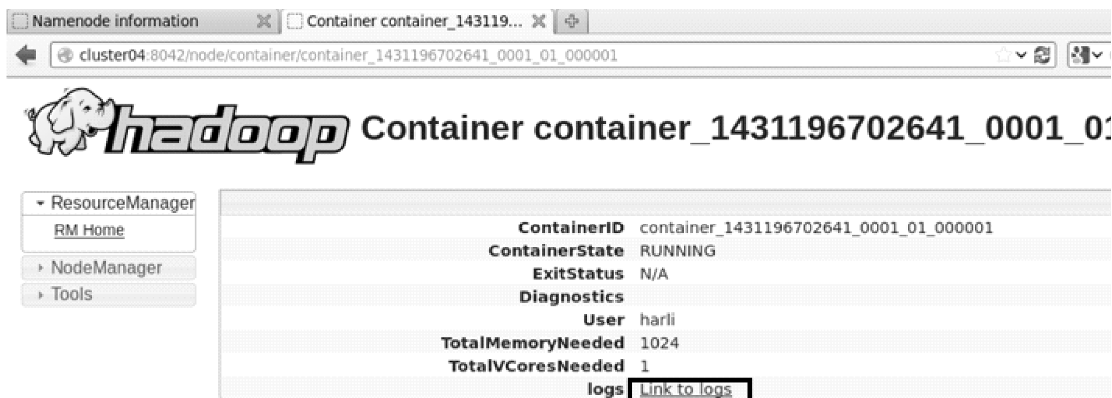


图 4.28　HadoopRM 的指定应用的容器的日志信息

6）单击 Link to logs，选择特定日志信息，如图 4.29 所示。



图 4.29　HadoopRM 的指定应用的容器的日志信息

7）单击 stdout:Total file length is 45405 bytes，查看 stdout 日志信息，如图 4.30 所示。

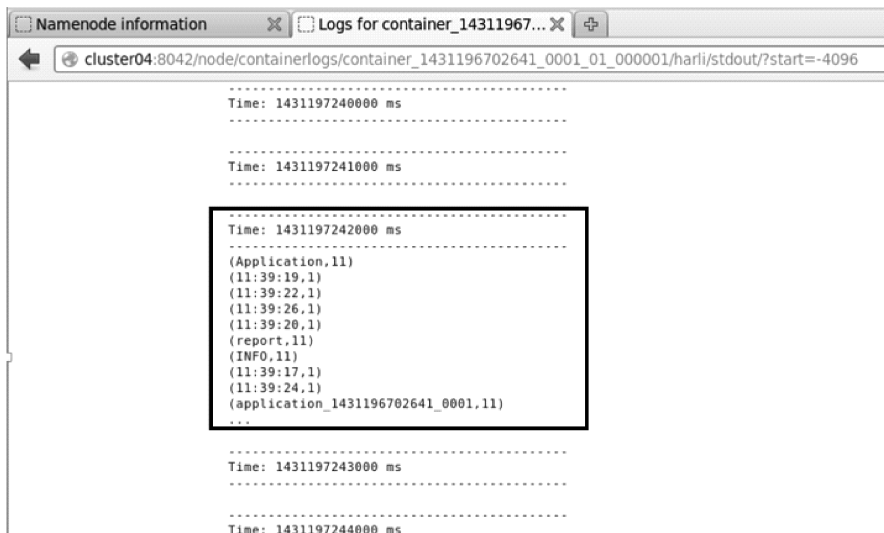


图 4.30　HadoopRM 的指定应用的容器的 stdout 日志信息

可以看到，stdout 中已经成功输出在 client 模式提交时的界面信息。

4.3.2 处理 HDFS 文件数据源的案例与解析

除了 Sockets，StreamingContext API 还提供了从其他基础数据源创建 DStream 实例的方法，这里以文件数据源作为例子，解析文件流的处理，并在此基础上，引入 Spark SQL，结合 Spark Streaming 和 Spark SQL 给出案例。

当企业的数据从各种数据源获取后，存入某个文件存储系统时（一般使用 HDFS），比如将从 Flume 数据源收集来的日志文件存入 HDFS 文件系统等，可以使用文件流的方式去处理，该方法可以监控某一目录下的创建文件，并对文件进行处理。

案例中以手动构建文件，并移入监控目录来简化外部数据源存入该目录。实际在企业中应用时，应该引入类似 Flume 等日志聚合系统负责数据收集。

这里换一种方式执行应用，即在 IDEA 中运行应用程序，使用 local 方式运行，这种方式下，可以方便代码的调试。

具体代码如下：

```
package stream
import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.sql.SQLContext
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.{Seconds,StreamingContext}

/**
  * Created by lenovo on 2015/4/13.
  */

//这里使用 lazy 加载的单件模式(singleton pattern)的方式来构建 SQLContext 实例
//可以避免在 foreachRDD 中重复构建
    objectSQLContextSingleton {
  @transient private var instance:SQLContext = null

  //lazy 方式实例化
  def getInstance(sparkContext:SparkContext):SQLContext = synchronized {
    if (instance == null) {
      instance = new SQLContext(sparkContext)
    }
    instance
  }
}
//样本类,用于构建 RDD 对应的 DataFrame 实例
//可以根据实际的数据格式,给出对应的解析样本类
case class Row(word:String)
object HdfsApp {
  def main(args:Array[String]) {
  //减少控制台输出信息
```

```
    import org.apache.log4j.{Level,Logger}
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.streaming").setLevel(Level.WARN)
    //日志文件监控路径
val logFile = args(0)
    val conf = new SparkConf().setAppName("HdfsApp").setMaster("local[1]")
    val ssc = new StreamingContext(conf,Seconds(1))
    //在 streaming 的内部处理中,使用 DataFrame,将读取到的文件数据
    //注册到临时表中,并将表查询结果显示到控制台上
    val words:DStream[String] = ssc.textFileStream()
    //这里使用 foreachRDD,针对每个 RDD 进行 Spark SQL 操作
    words.foreachRDD { rdd =>
if(!rdd.isEmpty()) {
        //获取 SQLContext 单例
        val sqlContext = SQLContextSingleton.getInstance(rdd.sparkContext)
        //代码中需要手动添加以下的隐式转换
        import sqlContext.implicits._
        //将 RDD 转换为 DataFrame
        val wordsDataFrame = rdd.map(w => Row(w)).toDF()
        //将转换结果注册成临时表
wordsDataFrame.registerTempTable("words")
        //对临时表执行 sql 语句
        val wordCountsDataFrame =
sqlContext.sql("select word,count(*) as total from words group by word")
wordCountsDataFrame.show()
      }
    }
ssc.start()
ssc.awaitTermination()
  }
}
```

打开配置 Run 的窗口，如图 4.31 所示。

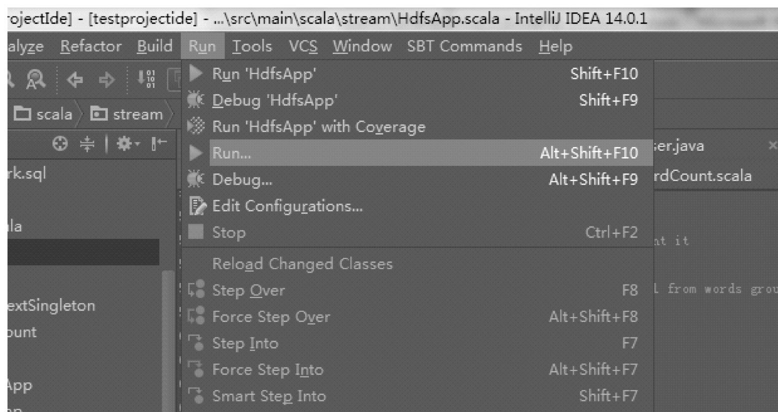


图 4.31　IDEA 的应用的 Run 配置

进入 Run 配置的编辑界面，如图 4.32 所示。

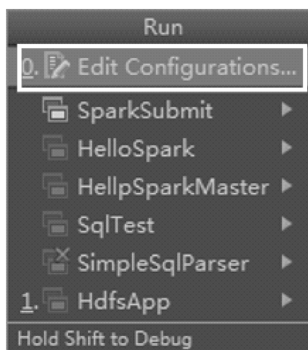


图 4.32　IDEA 的应用的 Run 配置的编辑菜单

输入具体配置信息，如图 4.33 所示。

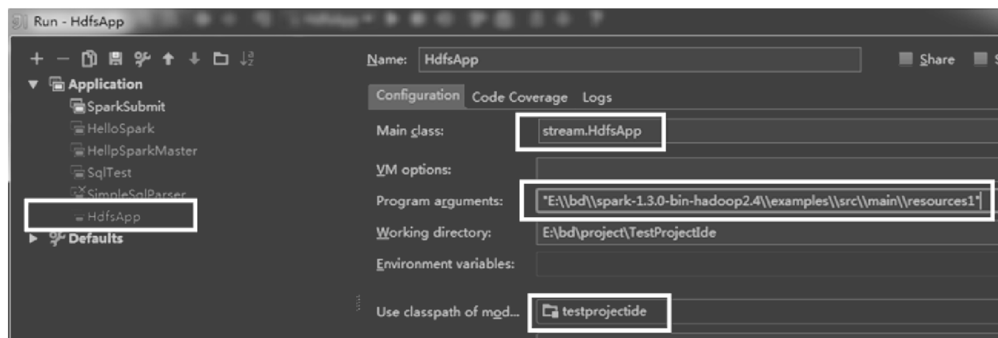


图 4.33　IDEA 的应用的 Run 配置的编辑界面

其中，在 Program arguments 部分，添加监控的路径，这里设置为本地文件系统下的监控目录“E:\\bd\\spark-1.3.0-bin-hadoop2.4\\examples\\src\\main\\resources1”。由于不是集群模式提交，因此 Main class 部分可以设置为当前的应用类，需要使用类的全路径。

单击窗口的 Run 按钮，启动流处理。然后，手动将文件添加到监控目录下。

注意：

1. 监控目录下的文件应该有一样的数据格式，避免在内部解析时报错。
2. 文件必须是在监控目录下创建，可以通过原子性的移动或重命名操作，放入目录。
3. 一旦移入目录，文件就不能再修改了，如果文件是持续写入的话，新的数据是无法读取的。

案例中，必须在启动后新建一个文件，然后移入目录，创建时间比启动早的文件，移入目录时不会处理。

案例中，使用的文件内容源自：E:\bd\spark-1.3.0-bin-hadoop2.4\examples\src\main\resources\kv1.txt，即 Spark 自带的 kv 文件。由于没有安装 HDFS，所以默认的是本地文件系统，不需要添加 file://的 scheme 信息。对应在 HDFS 系统上时，可以增加 hdfs://的 scheme 信息。

输出结果为：

```
15/04/13 01:56:01 INFO Server:jetty-8.y.z-SNAPSHOT
15/04/13 01:56:01 INFOAbstractConnector:Started SelectChannelConnector@0.0.0.0:4040
15/04/13 01:56:21 INFOFileInputFormat:Total input paths to process:1
word          total
494val_494 1
429val_429 2
230val_230 5
43val_43   1
282val_282 2
129val_129 2
217val_217 2
382val_382 2
470val_470 1
317val_317 2
457val_457 1
369val_369 3
145val_145 1
333val_333 2
95val_95   2
35val_35   3
65val_65   1
197val_197 2
485val_485 1
221val_221 2
```

Spark 1.3 版本中，增加了 EmptyRDD 的定义，用于源数据输入为空时构建的 RDD，这里的代码，添加了 EmptyRDD 的判断，即 if(!rdd.isEmpty()){…}，通过判断 RDD 是否为空，来过滤空数据，从而避免相应的 job 提交。添加判断后，输出界面会像上面那样，只有收到数据时，才会提交 job，进行处理。

如果没有添加该判断的话，代码会一直提交任务，但没有执行具体的数据处理，对应的界面如下：

```
15/04/13 01:58:58 INFOAbstractConnector:Started SelectChannelConnector@0.0.0.0:4040
word total
word total
word total
word total
word total
```

另外，在代码中，MasterURL 使用的是 .setMaster("local[1]")，因为文件流不需要 Receiver，也就不需要额外占用一个内核。

之前在 spark-shell 提交应用的方式下提到过 spark-shell 交互式中已经自动导入了 SQLContext 的隐式导入，因此不需要再自己添加，但对应的 spark-submit 方式提交应用时，必须手动在使用的代码中添加“import sqlContext.implicits._”这句隐式转换的导入语句，否则，后续的 toDF 等调用会编译失败。

在企业级的实时流处理中往往会引入 Kafka 作为分布式消息系统，以及 Flume 作为各种数据的收集系统。下面分别给出 Spark Streaming 整合 Kafka 的案例与解析，以及整合 Flume

的案例与解析。

4.3.3 处理 Kafka 数据源的准备工作



Kafka 是一种高吞吐量的分布式发布订阅消息系统，这里实现两种读取 Kafka 数据的方法：

1）一种是 Spark 1.3 版本之前的方法，使用 Receivers 和 Kafka 提供的高层次的 API。

2）一种是 Spark 1.3 引入的一个试验性的方法，方法中不再使用 Receivers，而是直接调用 Kafka 提供的低层次的 API。

这两种方法有着不同的编程模型、性能特征和语义级的保证，在读取数据的细节上有所不同。

一、Kafka 基础知识的准备

发布消息通常有两种模式：队列模式（Queuing）和发布－订阅模式（Publish－Subscribe）。队列模式中，Consumers 可以同时从服务端读取消息，每个消息只被其中一个 Consumer 读到；发布－订阅模式中消息被广播到所有的 Consumer 中。

Kafka 的 Topic 的分区数，是 Consumer 可以读取的并行数的最高限制值，这里对应 Spark Streaming 并行读取的最大值。

当 Consumer 使用相同的 groupId 去读取同一个 Topic 数据时，该 Topic 会将分区数据分发到各个 Consumer，即队列模式的消息发布模式，如果 Consumer 使用不同的 groupID 去读取同一个 Topic 数据时，该 Topic 的分区数据会广播到各个 Consumer 上，即使用广播的消息发布模式。

二、Kafka 集群的准备

为了简化 Kafka 集群的搭建，集中针对 Spark Streaming 对 Kafka 流数据处理的实践上，这里以尽可能简单地方式构建 Kafka 集群。这里使用 kafka_2.10－0.8.2.1.tgz 版本。

简单搭建步骤如下：

1. 获取 Kafka 部署包，并解压到指定目录

可以到 Kafka 的官方网站 http://kafka.apache.org/，下载部署包，这里使用 wget 命令下载，并解压：

```
[harli@wxx215 cluster_13]$wget http://apache.fayea.com/kafka/0.8.2.1/kafka_2.10-0.8.2.1.tgz
[harli@wxx215 cluster_13]$tar xvf /usr/harli/kafka_2.10-0.8.2.1.tgz
```

当前在 wxx 集群的 wxx215 节点部署 Kafka。

2. 启用默认配置的 Zookeeper 服务，可以直接使用现有的 Zookeeper 集群

```
bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
```

启动命令中－daemon 选项用户设置启动脚本在后台运行。启动后，使用 jps 命令查看进程，会看到 Zookeeper 服务：

```
[harli@wxx215 kafka_2.10-0.8.2.1]$bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
[harli@wxx215 kafka_2.10-0.8.2.1]$jps
```

```
26327 Worker
26156DataNode
7117QuorumPeerMain
25772NodeManager
7138 Jps
```

其中，QuorumPeerMain 对应启动的 Zookeeper 服务。

3. 启动 Kafka 服务

输入命令修改配置属性：

```
[harli@ wxx215 kafka_2.10 -0.8.2.1]$vim ./config/server.properties
```

修改相关的属性，当前只修改下面两个属性：

```
# Hostname the broker will bind to. If not set,the server will bind to all interfaces
#host.name = localhost
host.name = wxx215
#zookeeper.connect = loCalhost:2181
  zookeeper.connect = wxx215:2181
```

当前 Zookeeper 和 Broker 都在 wxx215 机器上启动，后续可以在其他机器上添加 Kafka 的服务。

服务属性文件中：

```
broker.id = 0
port = 9092
log.dir = /tmp/kafka - logs
```

其中：

1）broker.id 属性：配置信息是服务的全局唯一标识，当前为第一个服务，因此直接使用，不做修改，整个 Kafka 中服务的 broker.id 值必须唯一不能重复。

2）port 属性：服务使用的端口号，如果是在单台机器上启动多个 broker 服务，那么需要使用不同的端口号。

3）log.dir 属性：用于 Kafka 记录日志文件的目录，如果在单台机器上启动多个 broker 服务的话，应该设置成不同目录，避免多个 broker 服务在相同目录下生成目录文件。

修改完服务的属性文件后，启动服务：

```
bin/kafka - server - start.sh - daemon config/server.properties
```

其中，-daemon 选项指定后台方式运行。

使用 jps 命令查看：

```
[harli@ wxx215 kafka_2.10 -0.8.2.1]$bin/kafka - server - start.sh - daemon config/server.properties
[harli@ wxx215 kafka_2.10 -0.8.2.1]$jps
26327 Worker
7412Kafka
26156DataNode
7117QuorumPeerMain
25772NodeManager
```

```
7456 Jps
```

其中，Kafka 就是启动的 broker 服务的进程。

假设重新启动一个新的 broker，复制 config/server. properties 为 config/server_1. properties，修改其中关键的三个属性为：



```
broker. id = 1
port = 9093
log. dir = /tmp/kafka - logs - 1
```

启动 broker 服务：

```
bin/kafka - server - start. sh - daemon config/server_1. properties
```

对应的进程如下：

```
[harli@ wxx215 kafka_2. 10 - 0. 8. 2. 1] $ bin/kafka - server - start. sh - daemon config/server
_1. properties
[harli@ wxx215 kafka_2. 10 - 0. 8. 2. 1] $jps
26327 Worker
7585 Jps
7535Kafka
7412Kafka
26156DataNode
7117QuorumPeerMain
25772NodeManager
```

如果停止服务可以启动 bin/kafka - server - stop. sh 或直接 kill - 9 pid 方式，但是，脚本方式会 kill 掉当前所有的 Kafka 服务（具体可以查看脚本命令），因此如果在单机上启动了多个服务，而只需要停止其中某一个时，应该选用 kill 命令。

创建 Kafka 的 Topic，为了简化，这里使用一个 Topic，输入创建命令：

```
[harli@ wxx215 kafka_2. 10 - 0. 8. 2. 1] $bin/kafka - topics. sh -- create -- zookeeper wxx215:2181 -
- replication - factor 2 -- partitions 4 -- topic kafka_test
Created topic "kafka_test".
```

创建名为 kafka_test 的 topic，复制因子设为 2，同时分区数为 4，注意，分区数是 read parallelisms 的最大值。

查询 Kafka 当前的 Topic 信息，输入命令：

```
[harli@ wxx215 kafka_2. 10 - 0. 8. 2. 1] $bin/kafka - topics. sh -- list -- zookeeper wxx215: 2181
kafka_test
```

指定 -- zookeeper 选项的值为 wxx215：2181，对应的 Topic，即刚创建的。

创建 Producer 的提交脚本 start - producer. sh：

```
#!/usr/bin/env bash
./bin/spark - submit   -- master spark://192. 168. 70. 214: 7077 \
                       -- deploy - mode client \
                       -- driver - memory 1g \
                       -- driver - cores 1 \
```

```
--total-executor-cores 3 \
--executor-memory 1g \
--class stream.KafkaWordCountProducer \
--jars ./lib/spark-examples-1.3.0-hadoop2.4.0.jar \
../applications/testprojectide.jar wxx215:2181 \
kafka_test 20 10
```

该脚本对应的类的使用方法：

```
Usage: KafkaWordCountProducer <metadataBrokerList> <topic>" + " <messagesPerSec> <words-
PerMessage>
```

其中，参数 metadataBrokerList 的值为：wxx215:9092,wxx215:9093，即当前启动的 Kafka 服务（broker 列表，逗号分隔）；参数 topic 的值即刚才创建的 Topic 的名字 kafka_test；参数 messagesPerSec 的值为 20，即每个间隔时间发送的消息条数；参数 wordsPerMessage 的值为 10，即每条消息中的单词个数。

创建 Consumer 的提交脚本 start-consumer.sh：

```
./bin/spark-submit  --master spark://192.168.70.214:7077 \
                     --deploy-mode client \
                     --driver-memory 1g \
                     --driver-cores 1 \
                     --total-executor-cores 6 \
                     --executor-memory 1g \
                     --class stream.KafkaWordCount \
                     --jars ./lib/spark-examples-1.3.0-hadoop2.4.0.jar \
                    ../applications/testprojectide.jar wxx215:2181 \
                    group1 kafka_test 4
```

该脚本对应的类的使用方法：

```
Usage: KafkaWordCount <zkQuorum> <group> <topics> <numThreads>
```

其中，参数 zkQuorum 的值为：wxx215:2181，即当前启动的 Zookeeper 连接属性（Host:port 列表，逗号分隔）；参数 group 的值是指定当前 Consumer 的 groupId，这里设置为 group1；参数 topics 的值是 kafka_test，即刚才创建的 Topic 的名字 kafka_test；参数 numThreads 的值是 4，即读取 Kafka 流的线程数，当前设置成分区数的个数，对应的每个线程读取一个分区数据。

4.3.4 基于 Receiver 读取 Kafka 数据的案例与解析

基于 Receiver 读取的方法，使用了一个 Receiver 来获取数据，使用 Kafka 的高层次 API 实现了数据接收。在所有的 Receivers 中，通过一个 Receiver 从 Kafka 接收数据，并存入 Spark Executors，在 Spark Streaming 处理数据时提交 jobs。

由于使用了 Kafka 的高层次 API，数据读取的 offset 信息由 Spark 负责管理，一旦处理失败将会丢失数据，因此 Spark Streaming 引入了 WAL（Write Ahead Logs），用于保存 offset 信息，在失败时，重新读取 offset 信息，开始读取数据。

从高层次的角度看，之前的和 Kafka 集成方案使用 WAL 工作方式如下：

1）运行在 Spark Workers/Executors 上的 Kafka Receivers 连续不断地从 Kafka 中读取数据，其中用到了 Kafka 中高层次的消费者 API。

2）接收到的数据被存储在 Spark Workers/Executors 的内存中，同时也被写入到 WAL 中。只有接收到的数据被持久化到 Log 中，Kafka Receivers 才会去更新 Zookeeper 中 Kafka 的偏移量。

3）接收到的数据和 WAL 存储位置信息被可靠地存储，如果期间出现故障，这些信息被用来从错误中恢复，并继续处理数据。

一、应用程序准备

这里以 Spark 提供的 example 案例作为基础，进行案例实践分析，代码如下：

```
package stream
import java.util.Properties
import kafka.producer._
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.SparkConf
import org.apache.spark.examples.streaming._

/**
  * Created by lenovo on 2015/4/12.
  */
object KafkaWordCount {
  def main(args:Array[String]) {
    if (args.length < 4) {
      System.err.println("Usage:KafkaWordCount <zkQuorum> <group> <topics> <numThreads>")
      System.exit(1)
    }
    //除过多的日志信息,可以临时添加以下代码
    import org.apache.log4j.{Level,Logger}
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.streaming").setLevel(Level.WARN)
StreamingExamples.setStreamingLogLevels()
    //模式匹配,unapply 方法
    val Array(zkQuorum,group,topics,numThreads) = args
    val sparkConf = new SparkConf().setAppName("KafkaWordCount")
    val ssc = new StreamingContext(sparkConf,Seconds(2))
    //设置 checkpoint 目录,用于 Driver 的容错和对 RDD 的周期性 checkpoint
    ssc.checkpoint("checkpoint")
    //可以同时接收多个 topic 的数据,每个接收的线程数为 numThreads
      valtopicMap = topics.split(",").map((_,numThreads.toInt)).toMap
    val lines = KafkaUtils.createStream(ssc,zkQuorum,group,topicMap).map(_._2)
    val words = lines.flatMap(_.split(" "))
    //val wordCounts = words.map(x => (x,1L)).reduceByKeyAndWindow(_ + _,_ - _,Minutes
(10),Seconds(2),2)
    //val wordCounts = words.map(x => (x,1L)).reduceByKeyAndWindow(_ + _,_ - _,Minutes
```

```
(5),Minutes(3),2)
    val wordCounts = words.map(x => (x,1L)).reduceByKeyAndWindow(_ + _,_ - _,Minutes(1),
Minutes(1),2)
    wordCounts.print()
ssc.start()
ssc.awaitTermination()
  }
}
//这是 Kafka 的 Producer,即生产者,可以借鉴该方法,
//将 Spark Streaming 处理的数据写到 Kafka 中
//这里每隔 100ms 的时间生成指定范围的 int 作为 Work 向参数指定的 Topic 写数据
object KafkaWordCountProducer {
  def main(args:Array[String]) {
    if (args.length < 4) {
      System.err.println("Usage:KafkaWordCountProducer <metadataBrokerList> <topic>" +
        "<messagesPerSec> <wordsPerMessage>")
      System.exit(1)
    }
    val Array(brokers,topic,messagesPerSec,wordsPerMessage) = args
//Zookeeper 的连接属性
    val props = new Properties()
    props.put("metadata.broker.list",brokers)
    props.put("serializer.class","kafka.serializer.StringEncoder")
    val config = new ProducerConfig(props)
    val producer = new Producer[String,String](config)
    //向 topic 发送消息
    while (true) {
      val messages = (1 to messagesPerSec.toInt).map { messageNum =>
        val str = (1 to wordsPerMessage.toInt).map(x => scala.util.Random.nextInt(30).toString)
          .mkString(" ")
        new KeyedMessage[String,String](topic,str)
      }.toArray
      producer.send(messages:_*)
      Thread.sleep(100)
    }
  }
```

KafkaUtils.createStream 通过源码查看的话，可以看到内部默认设置了一些参数，

```
defcreateStream(
ssc:StreamingContext,
zkQuorum:String,
groupId:String,
    topics:Map[String,Int],
storageLevel:StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
  ):ReceiverInputDStream[(String,String)] = {
  val kafkaParams = Map[String,String](
    "zookeeper.connect" -> zkQuorum,"group.id" -> groupId,
    "zookeeper.connection.timeout.ms" -> "10000")
```

```
createStream[String,String,StringDecoder,StringDecoder](
ssc,kafkaParams,topics,storageLevel)
}
```

这里默认使用的存储级别为 StorageLevel. MEMORY_AND_DISK_SER_2，通过辅助来提供容错性，和 RDD 的默认存储级别不同；默认用于序列化的解码类为 StringDecoder，和 KafkaWordCountProducer 发送消息时的编码类是一样的。

注意：收发双方的编码应保持一致，避免出现乱码，有特殊需求时，可以自定义编解码的类。

二、无 WAL 的案例测试

启动 Producer 脚本 start－producer. sh 和 Consumer 脚本：start－consumer. sh。

当代码为：val wordCounts = words. map(x => (x,1L)). reduceByKeyAndWindow(_ + _,_ - _, Minutes(10),Seconds(2),2)时，在 start－consumer. sh 启动终端，每隔 2 秒就统计前 10 分钟的单词统计信息，其中 10 分钟为窗口长度，2 秒为批数据时间间隔。

下面是其中的一部分输出（当前仅一个 Consumer，分组为：Group 1 ，Topic 及其分区数为 Kafka_ test－ 4 partitions）：

```
-------------------------------------------
Time:1428911400000 ms
-------------------------------------------
(4,46425)
(8,46201)
(6,46325)
(0,46119)
(2,46068)
(7,46380)
(5,46464)
(9,46420)
(3,46019)
(1,45979)

-------------------------------------------
Time:1428911402000 ms
-------------------------------------------
(4,46815)
(8,46616)
(6,46741)
(0,46545)
(2,46471)
(7,46749)
(5,46856)
(9,46833)
(3,46413)
(1,46361)

-------------------------------------------
Time:1428911404000 ms
-------------------------------------------
```

```
(4,47207)
(8,46993)
(6,47109)
(0,46903)
(2,46848)
(7,47135)
(5,47218)
(9,47235)
(3,46799)
(1,46753)
```

为了深入分析 reduceByKeyAndWindow 方法的作用，这里使用 Kafka 自带的 console 的 Producer 来提供数据，对应执行命令为：

```
[harli@ wxx215 spark - 1.3.0 - bin - hadoop2.4]$../kafka_2.10 - 0.8.2.1/bin/kafka - console -
producer.sh -- broker - list wxx215:9092,wxx215:9093 -- topic kafka_test
```

测试滑动长度与输出的关系，使用的代码：val wordCounts = words.map(x => (x,1L)).reduceByKeyAndWindow(_ + _,_ - _,Minutes(5),Minutes(3),2)，即修改了时间，窗口长度为 5 分钟，滑动时间间隔为 3 分钟，对应的测试结果：

Consumer 的启动时间是 28 分钟 11 秒，打印时间整理见表 4.1。

表 4.1　打印结果的时间表

时　　间	输　　出
31 分钟	打印第一次
34 分钟	打印第二次
37 分钟	打印第三次

这里可以看到 reduceByKeyAndWindow 方法，滑动长度的 Minutes(3)值，对应了窗口数据统计的频率。对应的窗口长度，也可以通过这种方式去测试。把后一次的打印时间减去上一次的打印时间就是对应滑块长度 Minutes(3)，也就是窗口数据统计的频率。

这是 Consumer 界面对应的输出信息：

```
[Stage 0:==============================================================
=============================================================>
(46 +4) / 50]
-------------------------------------------
Time:1428913872000 ms
-------------------------------------------
(4,6)
(2,12)
(,3)
(11,1)
(3,12)
(1,41)
-------------------------------------------
Time:1428914052000 ms
```



```
-------------------------------------------
(4,0)
(2,102)
(,6)
(11,1)
(3,0)
(1,31)
-------------------------------------------
Time:1428914232000 ms
-------------------------------------------
(4,0)
(2,72)
(,0)
(11,0)
(3,0)
(1,0)
-------------------------------------------
Time:1428914412000 ms
-------------------------------------------
(4,0)
(2,0)
(,0)
(11,0)
(3,0)
(1,0)
```

可以看到，前面在 Producer 界面输入数据后，每隔滑动时间 3 分钟就统计一次前窗口长度 5 分钟的单词信息，到后面，Producer 不再输入数据，统计信息也就变成了 0。

在实时流处理中，窗口的概念是很重要的，经过上面这种人为地控制输入数据的测试，这里进一步对 reduceByKeyAndWindow 方法进行详细分析。

reduceByKeyAndWindow 有两个重载方法，如下面两种样例：

1）valwordCounts = words. map (x => (x,1 L)). reduceByKeyAndWindow (_ + _, Seconds (10s),seconds(4))。

2）valwordCounts = words. map (x => (x,1L)). reduceByKeyAndWindow (_ + _,_ - _, Seconds(10s),seconds(4),2)。

以 KafkaWordCount 这个 Consumer 为例，上面样例对应的窗口操作可以理解成，每隔 4 秒的时间，统计一次过去 10 秒的时间内的所有输入数据的单词统计信息。这里的窗口长度 Seconds(10s)，表示统计的范围，滑动时间间隔 seconds(4)表示每隔 4 秒就开始统计一次。

这是官方网站上对窗口方法相关概念的图形描述，这里在原图上加了一些细节说明，如图 4. 34 所示。这些说明中的一些概念可以参考章节 4. 2 Spark Streaming 基础概念部分。

两个长方形框，描述一个窗口信息，其中包含了 3 个时间片的批数据，对应的时间跨度就是“3 乘时间片”；滑动时间间隔，是指从第一个窗口滑动（沿着时间流方向前进）到第二个窗口时，经过的时间片，由于是 2 个时间片，对应的滑动时间间隔就是“2 乘时间片”。

reduceByKeyAndWindow 方法也是窗口方法之一，其他的窗口方法，在理解窗口长度和

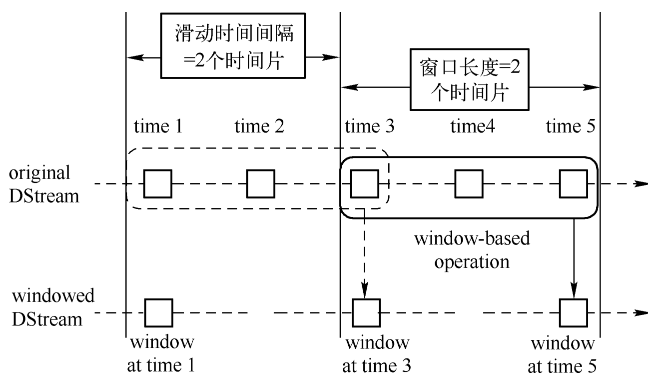


图 4.34　窗口长度与滑动时间间隔图

滑动时间间隔之后都比较好理解，这里针对相对复杂的 reduceByKeyAndWindow 方法进行深入分析，加强对窗口类方法的理解。

这里通过图形来描述 reduceByKeyAndWindow 的重载方法的异同点，如图 4.35 所示。

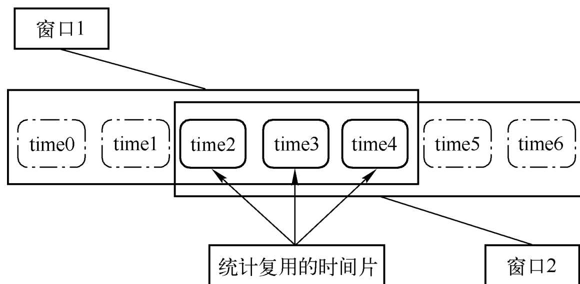


图 4.35　reduceByKeyAndWindow 方法描述图

对应的方法调用为 reduceByKeyAndWindow(_ + _, Seconds(5s), seconds(2))。该方法在 Scala API 官方网站上的定义如下：

> def reduceByKeyAndWindow(reduceFunc:(V,V)⇒V, invReduceFunc:(V,V)⇒V, windowDuration:Duration, slideDuration:Duration, partitioner:Partitioner, filterFunc:((K,V))⇒Boolean):DStream[(K,V)]
> Return a newDStream by applying incremental reduceByKey over a sliding window.
> def reduceByKeyAndWindow(reduceFunc:(V,V)⇒V, windowDuration:Duration):DStream[(K,V)]
> Return a newDStream by applying reduceByKey over a sliding window on this DStream.

其他的重载方法只是细节上的差异，这里不做具体分析。下面开始分析该 API。

1. def reduceByKeyAndWindow(reduceFunc:(V,V)⇒V, windowDuration…

以 reduceFunc:(V,V) 中 V 为"_ + _"为例进行分析。

该方法只是根据滑动时间间隔，对窗口长度内的时间片数据进行 reduceFunc 统计，对应的结果如下。

1）窗口 1 的统计结果：win1 = time0 + time1 + time2 + time3 + time4

2）窗口 2 的统计结果：win2 = time2 + time3 + time4 + time5 + time6

在图 4.35 中，对应为简单地各自统计自己的 5 个时间片的数据。

2. def reduceByKeyAndWindow(reduceFunc:(V,V)⇒V, invReduceFunc:(V,V), windowDu-

ration…

以 reduceFunc:(V,V)中 V 为“_+_”，invReduceFunc:(V,V)中 V 为“_-_”为例进行分析。



该方法复用了两个窗口中，时间片的交集部分的统计结果，即图中的时间片交集为 time2，time3，time4，因此进行 reduce 时，复用了这三个时间片的结果，对应的结果可以表示为：

1）窗口 1 的统计结果：win1 = time0 + time1 + **time2 + time3 + time4**

2）窗口 2 的统计结果：win2 = **win1** + time5 + time6 - **time0 - time1**

即，窗口 2 的统计是在窗口 1 的统计基础上，加上新增的时间片，同时减去移出的时间片。

根据上面的分析，我们可以知道，reduceFunc 和 invReduceFunc 这两个方法必须是互逆的，像“_+_”和“_-_”是互逆操作一样。

同时，还可以看出，当滑动时间间隔远小于窗口长度时，前后两个窗口的时间片交集就越多，因此使用第二种方法可以复用更多的时间片的统计结果，对应的性能也就更高了。如果滑动时间间隔等于或大于窗口长度时，由于没有可复用的时间片统计，这时候直接使用第一种方法就可以了。

三、窗口方法的源码解析

在 DStream 类中提供了 window 的方法，通过查看该方法的源码，进一步解析窗口操作。

```
def window(windowDuration:Duration,slideDuration:Duration):DStream[T] = {
  new WindowedDStream(this,windowDuration,slideDuration)
}
```

查看各种窗口相关的 API，基本上都是间接或直接调用了 window 的方法。

因此，通过对 window 方法，也就是对 WindowedDStream 类的深入源码解析，可以了解窗口操作的具体实现细节。

主构造函数：

```
private[streaming]
class WindowedDStream[T:ClassTag](
    parent:DStream[T],
    _windowDuration:Duration,
    _slideDuration:Duration)
  extends DStream[T](parent.ssc) {
……
```

这里父依赖 DStream 为传入的 DStream，在主构造函数体内：

```
if (!_windowDuration.isMultipleOf(parent.slideDuration)) {
  throw new Exception("The window duration of windowed DStream (" + _windowDuration + ") " + "
  must be a multiple of the slide duration of parent DStream (" + parent.slideDuration + ")")
}
if (!_slideDuration.isMultipleOf(parent.slideDuration)) {
  throw new Exception("The slide duration of windowed DStream (" + _slideDuration + ") " + "must be
  a multiple of the slide duration of parent DStream (" + parent.slideDuration + ")")
```

```
}
```

这里在构造 WindowedDStream 实例时，就对窗口长度和滑动时间间隔做了检查，两者都必须是 parent. slideDuration 的倍数，parent. slideDuration 对应的就是 DStream 的 slideDuration 属性，查看源码可知，该属性即时间片的时间值，DStream 类中该属性的源码如下：

```
/** DStream 产生一个 RDD 的批处理时间间隔 */
def slideDuration: Duration
```

对外部数据源来说，slideDuration 就是我们设置的批处理的时间间隔，每持续 slideDuration 时间，就开始构建一个 RDD。对应的 WindowedDStream，是每隔一个滑动时间间隔构建一个 RDD，slideDuration 就是窗口操作中的滑动时间间隔。虽然都是每隔 slideDuration 间隔就构建一个 RDD，但普通的 DStream 构建的 RDD 的数据是在一个 slideDuration 间隔内的，而如果是 WindowedDStream 的话，RDD 包含的数据是 windowDuration 间隔内的。

总之，slideDuration 是控制 RDD 构建频率的时间间隔。windowDuration 是控制 RDD 数据对应的时间长度。同时，普通 DStream 的 windowDuration（实际上没有这个属性）等于 slideDuration，即等于时间片大小。

下面这两个属性用于控制内存中持久化时，数据保留的时间：

```
override def parentRememberDuration: Duration = rememberDuration + windowDuration
```

其中 rememberDuration 为 DStream 属性：

```
private[streaming] var rememberDuration: Duration = null
```

最后，最关键的一个方法是 compute 方法，对应源码为：

```
override def compute(validTime: Time): Option[RDD[T]] = {
  val currentWindow = new Interval(validTime - windowDuration + parent.slideDuration, validTime)
  val rddsInWindow = parent.slice(currentWindow)
  val windowRDD = if (rddsInWindow.flatMap(_.partitioner).distinct.length == 1) {
logDebug("Using partition aware union for windowing at " + validTime)
    new PartitionerAwareUnionRDD(ssc.sc, rddsInWindow)
  } else {
logDebug("Using normal union for windowing at " + validTime)
    new UnionRDD(ssc.sc, rddsInWindow)
  }
  Some(windowRDD)
}
```

其中：

1）val currentWindow = new Interval(validTime - windowDuration + parent. slideDuration, validTime)：parent. slideDuration 是构建 RDD 对应的时间，“validTime - windowDuration”是计算回退 windowDuration（窗口长度）的时间长度，为了和 validTime 一样，需要调整 Interval 的起始点，如图 4. 36 所示。

2）val rddsInWindow = parent. slice(currentWindow)：parent 即 DStream，slice 方法构建对应 currentWindow 这个时间范围的 RDD。

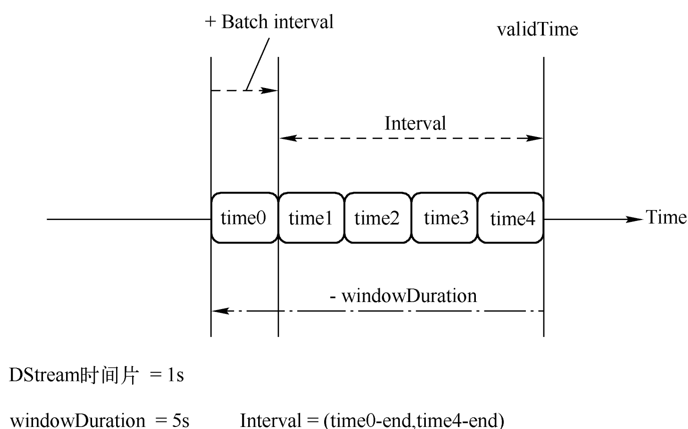


图 4.36 窗口操作中的 Interval 计算

3）最后通过 UnionRDD 操作将得到的 RDD 集合进行合并（RDD 实例是和时间片一一对应的，因此 slice 得到的应该是多个时间片的 RDD 的集合）。

由 UnionRDD 操作我们可以推导出，最终合并后的 RDD 的分区数将是批数据时间片对应的 RDD 的分区数的 sum（求和）值。因此，当分区数据量不是很大时，在进行窗口类型的操作时，可以优先考虑使用带分区设置参数的重载方法。相应的，内存中存储的 RDD 就应该足够装载指定窗口长度的对应数据量了。

底层的 RDD 是对应一个时间片的批数据的，对应的分区数应该是由该时间片内的数据设置。由于底层的参数 Spark. streaming. blockInterval 用于控制接收数据块的最小单位的时间，所以这个最小单位时间对应接收到的数据应该就是 RDD 中的 block。也就是说，batch-Interval/blockInterval 就是对应的 RDD 的分区数了。

比如，当前流的时间片设置为 2 秒时，由于参数 Spark. streaming. blockInterval 默认为 200 毫秒，因此对应的 RDD 的分区数是 10。

四、启动 WAL 的案例解析

为了避免在数据处理过程中出现故障而导致的数据丢失问题，在 Spark 1.2 中 Spark Streaming 引入了 WAL，用于保存 offset 信息，在失败时，可以通过重新加载 offset 信息，读取处理故障的数据，避免数据丢失。

启动 WAL 只需要配置属性“Spark. streaming. receiver. writeAheadLog. enable”，将默认的 false 修改为 true 即可。

该配置设置为 true 后，从一个 Receiver 接受的所有数据都会写入配置的 Checkpoint 目录中的一个预写日志（a write ahead log）中（可以参考 HBase 中的 WAL）。使用 WAL 可以避免在 Driver 恢复后的数据丢失问题，可以确保零数据丢失。由于增加了数据的预写日志，所以启动 WAL 不可避免地会降低每个 Receiver 的吞吐量。在实际应用时，我们可以通过使用更多的 Receiver 来并行接收数据，以提高整体的吞吐量。

在上面的源码分析中，我们已经看到了流的创建，对应创建创建流的源码如下：

```
def createStream(
ssc:StreamingContext,
zkQuorum:String,
```

```
groupId:String,
      topics:Map[String,Int],
storageLevel:StorageLevel = StorageLevel. MEMORY_AND_DISK_SER_2
   ):
```

创建流时，默认的 StorageLevel 为 StorageLevel. MEMORY_AND_DISK_SER_2，这里的_2 用于对数据进行备份，保证容错性，如果我们启动了 WAL，数据会自动备份到存储系统上。这时候可以将 input stream 默认的存储级别修改为 StorageLevel. MEMORY_AND_DISK_SER，即内部的数据存储不需要再进行数据备份了。

WAL 的启用，只需要修改一个配置，而同时在代码中构建流时，需修改默认的存储级别。修改配置属性可以在 Spark - defaults. conf 中增加以下一行内容：

```
Spark. streaming. receiver. writeAheadLog. enable true
```

注意：由于 WAL 要将数据存储到 Checkpoint 目录下，代码中也要调用 streamingContext. checkpoint( ) 来设置 Checkpoint 目录才可以。

这里有一个细节需要注意，即 Driver 失败后重启部分，如果是 Spark on Yarn 模式的话，由 Yarn 负责重启 Driver，即 ApplicationsMaster；如果是在 Standalone 模式下开启 Driver 失败时启动的话，可以查看 spark - shell 或 spark - submit 的帮助信息，即将 Driver 失败时重启的选项打开即可，如下所示：

```
spark standalone with cluster deploy mode only:
  --driver-cores NUM          Cores for driver (Default:1).
 --supervise                  If given,restarts the driver on failure.
  --kill SUBMISSION_ID        If given,kills the driver specified.
  --status SUBMISSION_ID      If given,requests the status of the driver specified.
```

在启动交互式或直接提交应用时，加上 -- supervise 就可以让 Driver 在失败时重启了，需要注意的是，这是在“Standalone + Cluster”模式下才有效的。

注意：WAL 在 Spark 1.3 版本中，Flume 部分在内部测试出现了回退现象，这部分已经提交了 PR，具体内容可以参考 Spark 1.3.1 版本的 release 说明。

由于本书重点内容在案例与解析部分，属性配置等相关内容介绍的不多，具体可以参考官方网站的配置页（http://Spark. apache. org/docs/latest/configuration. html）。

常见问题与分析：在使用 Receiver 接收数据的流处理中，有时候会碰到无法接收数据的现象，这时可以先从两个方面初步排查问题：

1）发布消息时，是否使用了队列模式的消息发布模式，这时候在某个 Consumer 上会表现为数据丢失等现象。

2）任何一个使用了 Receiver 的流处理，都需要注意一点，每个 Receiver 本身就会占用一个内核，如果没有分配足够的内核的话，就无法启动数据处理，表面上就像是没有接收到数据一样，实际上是没有启动数据处理任务。如果是 local 模式的话，对应地设置足够的 N(local[N])，如果是集群模式，对应的 Executor（一个 Executor 对应一个 Receiver）需要设置足够的内核个数。

Kafka 相关问题的分析与排查等，可以借助 Spark 应用程序的 Web Interface 界面上新增的 Streaming 页面，也可以借助第三方工具，如 ApacheKafka 监控系列 KafkaOffsetMonitor 开

源项目，可以用来监控 Kafka 的 Consumer 相关信息，来帮助分析与排查 Kafka 相关问题。

4.3.5 直接读取(无 Receiver)Kafka 数据的案例与解析

在 Spark 1.3 中，针对 Kafka 数据源引入的新的不借助 Receiver，而采用直接读取数据的方法，这个方法具有更强的端到端的保证。当启动处理数据的 Jobs 时，Kafka 底层的 Consumer API 会根据定义的 offset 范围从 Kafka 中读取数据。方法中通过周期性地查询每个“Topic + Partition”的最新 offsets，以及设置的 offset 范围，来获取批数据，进行处理。当前这一试验性的方法只在 Spark 1.3 的 Scala 和 Java API 中提供。

虽然，在 Spark 1.2 版本中，Spark Streaming 引入了 WAL，但是，如果 WAL 已经写成功了，但还没及时更新 Zookeeper 中的 Kafka 偏移量时，而 Kafka 是根据 Zookeeper 中记录的 Consumer 当前的 offset 来发送数据的，所以当系统出现故障时，还是会导致某些数据重复处理。

根本原因在于 Spark Streaming 的 WAL 操作和 Zookeeper 中的 offset 更新并不是一个原子性的操作，并不能保证 Spark Streaming 和 Zookeeper 中记录的数据 offset 的一致性。因此在 Spark 1.3 引入了 Direct API 概念，它可以在不使用 WAL 的情况下实现仅处理一次的语义(exactly - once semantics)。Direct API 直接使用 Kafka 的低层次 API，指定要读取的 offset。因此，真正使用的 offset 是由 Spark Streaming 通过 Checkpoints 来维护的，只在一个地方维护 offset，就不会出现不一致的问题了。

一、案例实践的准备

一般在企业中，会对一些日志信息、网络异常事件信息等进行统计，本案例针对这些信息，通过 Kafka 发送到 Spark Streaming，以关键字 ERROR 作为特征值，统计流持续的异常信息的条数，用两种不同的方式，统计从 Kafka 获取的数据的单词数量。

这里给出应用代码以及两种 offset 获取方式的实践案例与分析。

应用代码的如下：

```
package stream
import kafka.serializer.StringDecoder
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.SparkConf
import org.apache.spark.HashPartitioner
import org.apache.spark.examples.streaming._
/**
 * Consumes messages from one or more topics in Kafka and does wordcount.
 * Usage:DirectKafkaWordCount <brokers> <topics>
 *  <brokers> is a list of one or more Kafka brokers
 *  <topics> is a list of one or more kafka topics to consume from
 *
 * Example:
 * $bin/run-example streaming.DirectKafkaWordCount broker1-host:port,broker2-host:port \
 * topic1,topic2
 */
```

```
object DirectKafkaWordCount {
  def main(args:Array[String]) {
    if (args.length < 2) {
      System.err.println( s"""
        |Usage:DirectKafkaWordCount <brokers> <topics>
        |     <brokers> is a list of one or more Kafka brokers
        |     <topics> is a list of one or more kafka topics to consume from
        |
""".stripMargin)
      System.exit(1)
    }
    //屏蔽过多的日志信息
    import org.apache.log4j.{Level,Logger}
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.streaming").setLevel(Level.WARN)
    //默认情况下该句不起作用
StreamingExamples.setStreamingLogLevels()
    //模式匹配,从 args 中获取 brokers 和 topics
    val Array(brokers,topics) = args
    //以批数据时间片为 2 秒构建 StreamingContext 实例
    val sparkConf = new SparkConf().setAppName("DirectKafkaWordCount")
    val ssc = new StreamingContext(sparkConf,Seconds(10))
ssc.checkpoint("directcheckpoint")
    //使用传入的 brokers and topics 构建 Kafka 流
    val topicsSet = topics.split(",").toSet
val kafkaParams = Map[String,String]("metadata.broker.list" -> brokers)
    //这里添加一个 Kafka 的属性配置,设置后,会重置 offset,也就是会从 Kafka 那从头开始读取数据
//val kafkaParams = Map[String,String]("metadata.broker.list" -> brokers,"auto.offset.reset" -> "
smallest" )
     val messages = KafkaUtils.createDirectStream[String,String,StringDecoder,StringDecoder](
ssc,kafkaParams,topicsSet)
    //Get the lines,split them into words,count the error line ,words ,etc. and print
    val lines = messages.map(_._2)
//设置初始状态统计值
    val initialRDD = ssc.sparkContext.parallelize(List(("ERROR",0)))
    val errorWords = lines.filter(_.contains("ERROR")).map { x => ("ERROR",1) }
//val runningCounts = errorWords.updateStateByKey(updateErrorCount)
    valrunningCounts = errorWords.updateStateByKey[Int](newUpdateFunc,
      new HashPartitioner(ssc.sparkContext.defaultParallelism),true,initialRDD)
    //这是随着流计算的进行,不断更新的统计值
runningCounts.print()
    //这是流的单词统计,对应的 reduceByKey 实际是作用在其底层的 RDD 序列上
    //可以看到,基本上 DStream 提供的高层 API,作用和 RDD 的同名 API 基本是差不多的
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x,1L)).reduceByKey(_ + _)
wordCounts.print()
    //这里还可以使用 DStream 的 transform ,直接对 DStream 的底层 RDD 序列进行 reduceByKey 操作
    val wordCounts2 = words.map(x => (x,1L)).transform(_.reduceByKey(_ + _))
```

```
wordCounts2. print()
    //启动流处理
ssc. start()
ssc. awaitTermination()
  }
//流的状态更新函数,元素类型为 Key - Value 格式
//runningCount 是历史的状态值
//newValues 是新的流数据的 Value 序列值
//利用该序列值对 runningCount 进行更新,并返回更新后的最新状态值
def updateErrorCount(newValues:Seq[Int],runningCount:Option[Int]):Option[Int] = {
    //对新进来的值集合进行求和
    val currentCount = newValues. sum
    //获取当前的统计值——即状态值
    val previousCount = runningCount. getOrElse(0)
    //更新最新的统计值
    Some(currentCount + previousCount)
  }
//
  val newUpdateFunc = (iterator:Iterator[(String,Seq[Int],Option[Int])]) => {
iterator. flatMap(t => updateErrorCount(t. _2,t. _3). map(s => (t. _1,s)))
  }
}
```



这里为了测试方便，将流的批数据时间片设成了 10 秒。

在另一个终端或节点上，启动 Producer，对应脚本如下（或者用前面的 start - producer. sh）：

```
bin/kafka - console - producer. sh -- broker - listwxx215:9092,wxx215:9093 -- topic kafka_test
```

在 Producer 生产者界面上输入以下日志信息：

```
[2015 - 04 - 14 12:01:36,625] TRACE [Controller 0]:checking need to trigger partition rebalance (kafka. controller. KafkaController)
[2015 - 04 - 14 12:01:36,627] ERROR [Controller 0]:preferred replicas by broker Map(1 -> Map ([kafka_test,1] -> List(1,0),[kafka_test,3] -> List(1,0),[test,0] -> List(1)),0 -> Map([kafka_test,0] -> List(0,1),[kafka_test,2] -> List(0,1))) (kafka. controller. KafkaController)
[2015 - 04 - 14 12:01:36,627] ERROR [Controller 0]: topics not in preferred replica Map() (kafka. controller. KafkaController)
[2015 - 04 - 14 12:01:36,627] TRACE [Controller 0]:leader imbalance ratio for broker 1 is 0. 000000 (kafka. controller. KafkaController)
[2015 - 04 - 14 12:01:36,627] ERROR [Controller 0]: topics not in preferred replica Map() (kafka. controller. KafkaController)
[2015 - 04 - 14 12:01:36,627] TRACE [Controller 0]:leader imbalance ratio for broker 0 is 0. 000000 (kafka. controller. KafkaController)
[2015 - 04 - 14 12:06:36,625] TRACE [Controller 0]:checking need to trigger partition rebalance (kafka. controller. KafkaController)
[2015 - 04 - 14 12:06:36,626] ERROR [Controller 0]:preferred replicas by broker Map(1 -> Map ([kafka_test,1] -> List(1,0),[kafka_test,3] -> List(1,0),[test,0] -> List(1)),0 -> Map([kafka_test,0] -> List(0,1),[kafka_test,2] -> List(0,1))) (kafka. controller. KafkaController)
```

```
[2015 - 04 - 14 12:06:36,626] ERROR [Controller 0]: topics not in preferred replica Map()
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:06:36,626] TRACE [Controller 0]: leader imbalance ratio for broker 1 is 0.000000
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:06:36,626] ERROR [Controller 0]: topics not in preferred replica Map()
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:06:36,626] TRACE [Controller 0]: leader imbalance ratio for broker 0 is 0.000000
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:11:36,625] TRACE [Controller 0]: checking need to trigger partition rebalance
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:11:36,626] ERROR [Controller 0]: preferred replicas by broker Map(1 -> Map
([kafka_test,1] -> List(1,0),[kafka_test,3] -> List(1,0),[test,0] -> List(1)),0 -> Map([kafka_
test,0] -> List(0,1),[kafka_test,2] -> List(0,1))) (kafka.controller.KafkaController)
[2015 - 04 - 14 12:11:36,626] ERROR [Controller 0]: topics not in preferred replica Map()
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:11:36,626] TRACE [Controller 0]: leader imbalance ratio for broker 1 is 0.000000
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:11:36,627] ERROR [Controller 0]: topics not in preferred replica Map()
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:11:36,627] TRACE [Controller 0]: leader imbalance ratio for broker 0 is 0.000000
(kafka.controller.KafkaController)
[2015 - 04 - 14 12:11:36,627] ERROR [Controller 0]: topics not in preferred replica Map()
(kafka.controller.KafkaController)
```

这是从 Kafka 日志里截取的一段，将 INFO 信息替换为 ERROR 进行测试。实际企业环境中，可以通过 Flume - ng 收集日志信息，存入 Kafka，也可以自己编写 Kafka 的 Producer 客户端，向 Kafka 指定 Topic 发送消息等。

二、从 Topic 最新的 offset 开始读取数据的案例与分析

通过设置 Kafka 的参数为：

```
valkafkaParams = Map[String,String]("metadata.broker.list" -> brokers)
```

准备好日志数据之后，开始启动作为启动生产者的脚本：

```
[harli@wxx215 spark - 1.3.0 - bin - hadoop2.4]$./app/start - DirectKafkaWordCount.sh
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
```

界面每隔时间片长度 10 秒时间就打印一次统计信息，第一次 10 秒的统计信息如下：

```
-------------------------------------------
Time:1428990030000 ms
-------------------------------------------
(ERROR,10)
-------------------------------------------
Time:1428990030000 ms
-------------------------------------------
(replica,7)
(1),,3)
(checking,3)
(is,6)
```



```
(partition,3)
(12:06:36,625],1)
(ERROR,10)
(need,3)
([kafka_test,3],3)
(0,6)
…
-------------------------------------------
Time:1428990030000 ms
-------------------------------------------
(replica,7)
(1),,3)
(ERROR,10)
([kafka_test,3],3)
(0,6)
(12:11:36,625],1)
(trigger,3)
(topics,7)
(1))),3)
([test,0],3)
…
```

生产者界面一次输入包含 10 条 ERROR 的日志信息，可以看到，过滤 ERROR 并持续统计的结果是当前流一共有 10 条信息包含 ERROR 信息。

后面两个统计结果，只是使用了不同的方法实现相同的单词统计功能，因此显示的统计结果是一样的。

再次向生产者输入信息，这次先输入包含 9 条 ERROR 的日志信息，可以看到，过滤 ERROR 并持续统计的结果，是流数据的前一次统计状态值（10 条 ERROR 信息），加上新增的 9 条，一共包含 19 条 ERROR 的信息，信息如下：

```
-------------------------------------------
Time:1428990050000 ms
-------------------------------------------
(ERROR,19)
-------------------------------------------
Time:1428990050000 ms
-------------------------------------------
(replica,6)
(1),,3)
(checking,3)
(is,6)
(partition,3)
(12:06:36,625],1)
(ERROR,9)
(need,3)
([kafka_test,3],3)
(0,6)
…
```

```
-------------------------------------------
Time:1428990050000 ms
-------------------------------------------
(replica,6)
(1),,3)
(ERROR,9)
([kafka_test,3],3)
(0,6)
(12:11:36,625],1)
(trigger,3)
(topics,6)
(1))),3)
([test,0],3)
…
```

对应的最后一行数据在生产者界面输入后，对应的统计为：

```
-------------------------------------------
Time:1428990060000 ms
-------------------------------------------
(ERROR,20)
-------------------------------------------
Time:1428990060000 ms
-------------------------------------------
(replica,1)
(ERROR,1)
(topics,1)
(12:11:36,627],1)
([2015-04-14,1)
([Controller,1)
(not,1)
((kafka.controller.KafkaController),1)
(preferred,1)
(Map(),1)
…
-------------------------------------------
Time:1428990060000 ms
-------------------------------------------
(replica,1)
(ERROR,1)
(topics,1)
([2015-04-14,1)
((kafka.controller.KafkaController),1)
(preferred,1)
(Map(),1)
(in,1)
(0]:,1)
(12:11:36,627],1)
…
```

可以从单词统计中新增的 ERROR 单词数（这里日志的一条信息中只出现一次 ERROR），推出第一个统计结果确实保留了流数据的状态信息。

三、重置 offset，从 Topic 最老的 offset 开始读取数据的案例与分析



只需要在创建 Kafka 流时，将 Kafka 的参数 metadata. broker. list 设置为 smallest，就可以从 Kafka 中缓存的最早的数据开始读取。

通过设置 Kafka 的参数为：

```
//修改这里添加一个 Kafka 的属性配置,设置后,会重置 offset,也就是会从 Kafka 那从头开始读取数据
//val kafkaParams = Map[String,String]("metadata. broker. list" -> brokers,"auto. offset. reset" ->"
smallest" )
```

这里将注释去掉，同时注释掉上面一行的 kafkaParams 设置，然后启动应用，这时候不需要再在 Producer 生产者界面中输入任何信息，因为 Kafka 已经缓存了一批数据（默认缓存时间为 7 天）。

启动脚本：

```
[harli@ wxx215 spark - 1. 3. 0 - bin - hadoop2. 4]$. /app/start - DirectKafkaWordCount. sh
SparkSpark assembly has been built with Hive,including Datanucleus jars on classpath
```

每次启动后都会从 Kafka 的缓存中从头开始读取数据。对应界面提示信息中有：

```
15/04/14 14:12:54 INFOutils. VerifiableProperties:Verifying properties
15/04/14 14:12:54 INFOutils. VerifiableProperties:Property auto. offset. reset is overridden to smallest
15/04/14 14:12:54 INFOutils. VerifiableProperties:Property group. id is overridden to
15/04/14 14:12:54 INFOutils. VerifiableProperties:Property zookeeper. connect is overridden to
15/04/14 14:13:00 INFOutils. VerifiableProperties:Verifying properties
15/04/14 14:13:00 INFOutils. VerifiableProperties:Property auto. offset. reset is overridden to smallest
15/04/14 14:13:00 INFOutils. VerifiableProperties:Property group. id is overridden to
15/04/14 14:13:00 INFOutils. VerifiableProperties:Property zookeeper. connect is overridden to
15/04/14 14:13:00 WARN util. SizeEstimator:Failed to check whether UseCompressedOops is set; as-
suming yes
```

这是第一、二次启动脚本时的输出，每次都是从当前 Kafka 的缓存中重新读取，因此统计数据是一样的。

```
[harli@ wxx215 spark - 1. 3. 0 - bin - hadoop2. 4]$. /app/start - DirectKafkaWordCount. sh
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
15/04/14 14:10:58 WARN util. NativeCodeLoader:Unable to load native - hadoop library for your plat-
form…using builtin - java classes where applicable
15/04/14 14:10:59 INFO slf4j. Slf4jLogger:Slf4jLogger started
15/04/14 14:10:59 INFORemoting:Starting remoting
15/04/14 14:11:00 INFORemoting:Remoting started; listening on addresses:[akka. tcp://sparkDriver
@ wxx215:59763]
15/04/14 14:11:00 INFO server. Server:jetty - 8. y. z - SNAPSHOT
15/04/14 14:11:00 INFO server. AbstractConnector:Started SocketConnector@ 0. 0. 0. 0:51482
15/04/14 14:11:00 INFO server. Server:jetty - 8. y. z - SNAPSHOT
15/04/14 14:11:00 INFO server. AbstractConnector:Started SelectChannelConnector@ 0. 0. 0. 0:4040
15/04/14 14:11:02 INFOutils. VerifiableProperties:Verifying properties
15/04/14 14:11:02 INFOutils. VerifiableProperties:Property auto. offset. reset is overridden to smallest
```

```
15/04/14 14:11:02 INFOutils.VerifiableProperties:Property group.id is overridden to
15/04/14 14:11:02 INFOutils.VerifiableProperties:Property zookeeper.connect is overridden to
15/04/14 14:11:10 INFOutils.VerifiableProperties:Verifying properties
15/04/14 14:11:10 INFOutils.VerifiableProperties:Property auto.offset.reset is overridden to smallest
15/04/14 14:11:10 INFOutils.VerifiableProperties:Property group.id is overridden to
15/04/14 14:11:10 INFOutils.VerifiableProperties:Property zookeeper.connect is overridden to
15/04/14 14:11:10 WARN util.SizeEstimator:Failed to check whether UseCompressedOops is set; assuming yes
-------------------------------------------
Time:1428991870000 ms
-------------------------------------------
(ERROR,68)
-------------------------------------------
Time:1428991870000 ms
-------------------------------------------
(273,200)
(499,200)
(419,200)
(600,200)
(253,200)
(282,200)
(82,200)
(592,200)
(332,200)
(345,200)
…
-------------------------------------------
Time:1428991870000 ms
-------------------------------------------
(275,200)
(499,200)
(419,200)
(488,200)
(118,200)
(202,200)
(282,200)
(378,200)
(547,200)
(426,200)
…
-------------------------------------------
Time:1428991880000 ms
-------------------------------------------
(ERROR,68)
-------------------------------------------
Time:1428991880000 ms
-------------------------------------------
-------------------------------------------
Time:1428991880000 ms
-------------------------------------------
```

这里再在 Kafka 生产者界面输入 10 条包含 ERROR 的消息，然后重新启动，预期结果应该仅仅将 ERROR 的状态计数统计更新为：（ERROR，68 + 10）



```
[harli@wxx215 spark-1.3.0-bin-hadoop2.4]$./app/start-DirectKafkaWordCount.sh
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
15/04/14 14:31:29 WARN util.NativeCodeLoader:Unable to load native-hadoop library for your platform…using builtin-java classes where applicable
15/04/14 14:31:30 INFO slf4j.Slf4jLogger:Slf4jLogger started
15/04/14 14:31:30 INFORemoting:Starting remoting
15/04/14 14:31:30 INFORemoting:Remoting started; listening on addresses:[akka.tcp://sparkDriver@wxx215:42711]
15/04/14 14:31:31 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/14 14:31:31 INFO server.AbstractConnector:Started SocketConnector@0.0.0.0:49258
15/04/14 14:31:36 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/14 14:31:36 INFO server.AbstractConnector:Started SelectChannelConnector@0.0.0.0:4040
15/04/14 14:31:38 INFOutils.VerifiableProperties:Verifying properties
15/04/14 14:31:38 INFOutils.VerifiableProperties:Property auto.offset.reset is overridden to smallest
15/04/14 14:31:38 INFOutils.VerifiableProperties:Property group.id is overridden to
15/04/14 14:31:38 INFOutils.VerifiableProperties:Property zookeeper.connect is overridden to
15/04/14 14:31:40 INFOutils.VerifiableProperties:Verifying properties
15/04/14 14:31:40 INFOutils.VerifiableProperties:Property auto.offset.reset is overridden to smallest
15/04/14 14:31:40 INFOutils.VerifiableProperties:Property group.id is overridden to
15/04/14 14:31:40 INFOutils.VerifiableProperties:Property zookeeper.connect is overridden to
15/04/14 14:31:40 WARN util.SizeEstimator:Failed to check whether UseCompressedOops is set; assuming yes
-------------------------------------------
Time:1428993100000 ms
-------------------------------------------
(ERROR,78)

-------------------------------------------
Time:1428993100000 ms
-------------------------------------------
(273,200)
(499,200)
(419,200)
(600,200)
(253,200)
(282,200)
(82,200)
(592,200)
(332,200)
(345,200)
…
```

如上面的(ERROR,78)就是预期的结果。

四、深入状态更新函数的源码解析

Spark Streaming API 中比较难理解的除了窗口类型的 API 之外，还有一个就是状态更新

类型的 API。

这里从最简单的状态更新类型的 API 开始分析，案例中给出了两个 API 的实例，调用代码如下：

```
val runningCounts = errorWords. updateStateByKey[ Int ]( newUpdateFunc,
    new HashPartitioner( ssc. sparkContext. defaultParallelism) ,true,initialRDD)
```

这两者是一样的，为了更深入理解 updateStateByKey，先详细分析 updateErrorCount 和 newUpdateFunc 两个函数。分析函数有个比较简单的方法，可以启动 Scala 运行环境，然后直接将函数定义复制进去，这时候就可以看到交互式界面反馈的详细信息了，步骤如下：

首先要在本地安装 Scala 环境，具体安装方法可以从网上搜下，这里以 Windows 7 下为例，启动 Windows 下的 command 窗口，如图 4. 37 所示。

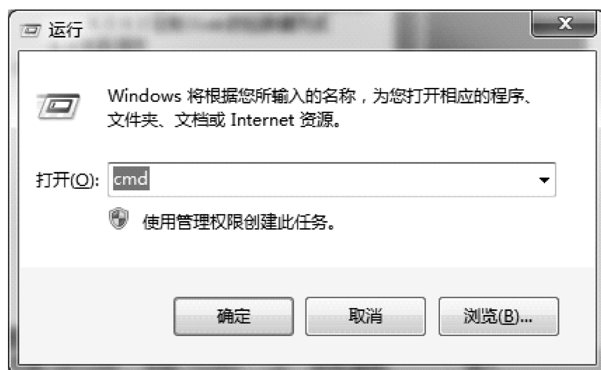


图 4. 37　启动 Windows 下的 command 窗口

打开 command 窗口后，输入 scala，按〈Enter〉键后出现 scala 交互式界面，如图 4. 38 所示。



图 4. 38　启动 Windows 下 scala 交互式界面

然后将代码复制进去，交互式界面的反馈信息如下：

```
scala >    def updateErrorCount( newValues:Seq[ Int ] ,runningCount:Option[ Int ] ) :Option[ Int ] = {
        |    val currentCount = newValues. sum
        |    val previousCount = runningCount. getOrElse(0)
        |    Some( currentCount + previousCount)
        |  }
updateErrorCount:( newValues:Seq[ Int ] ,runningCount:Option[ Int ] ) Option[ Int ]

scala >

scala >    val newUpdateFunc = ( iterator:Iterator[ ( String,Seq[ Int ] ,Option[ Int ] ) ] ) => {
```

```
|      iterator.flatMap(t => updateErrorCount(t._2,t._3).map(s => (t._1,s)))
|   }
new UpdateFunc:Iterator[(String,Seq[Int],Option[Int])] => Iterator[(String,Int)] = <function1 >
```

这里，函数字面量 newUpdateFunc 的类型是一个 <function1>，即带一个参数的 Function 类，输入参数的类型为 Iterator[(String, Seq[Int], Option[Int])]，返回类型为 Iterator[(String, Int)]。输入参数中，String 参数对应的是 Key，Seq[Int]对应的是 Value 序列，Option[Int]对应的是历史状态值；返回类型中的 String 对应 Key，而 Int 则是对应 Key 更新后的新的状态值。

接下来分析 updateStateByKey 方法的源码：

```
def updateStateByKey[S:ClassTag](
updateFunc:(Seq[V],Option[S]) => Option[S]
  ):DStream[(K,S)] = {
updateStateByKey(updateFunc,defaultPartitioner())
}
```

调用代码 updateStateByKey（updateErrorCount），这是最简单的状态更新 API 方式，只需要提供 Value 序列值和历史状态信息，就可得到新的状态信息的处理结果。内部使用的分区器是默认分区器，实际上就是 new HashPartitioner（ssc.sparkContext.defaultParallelism）构建的分区器，这部分可以参考章节 2.2.9 分区数设置的案例与源码解析部分。最终在各种默认参数被设置后，就到了第二个 API，及调用 updateStateByKey[Int](newUpdateFunc, new HashPartitioner(ssc.sparkContext.defaultParallelism), true, initialRDD)方法。最终构建了 StageDStream 实例：

```
def updateStateByKey[S:ClassTag](
updateFunc:(Iterator[(K,Seq[V],Option[S])]) => Iterator[(K,S)],
    partitioner:Partitioner,
rememberPartitioner:Boolean
  ):DStream[(K,S)] = {
    new StateDStream(self,ssc.sc.clean(updateFunc),partitioner,rememberPartitioner,None)
}
```

到此状态，更新类的 API 源码解析结束。对应构建的 StateDStream 源码解析可以参考前面的 WindowedDStream 源码解析。

4.3.6 处理 Flume 数据源的实践准备

Flume 是由 Cloudera 公司开发的一款高性能、高可能的分布式日志收集系统。Flume 的核心是把数据从数据源收集过来，再送到目的地。为了保证输送一定成功，在送到目的地之前，会先缓存数据，待数据真正到达目的地后，删除自己缓存的数据。

初始的 Flume 版本是 FlumeOG（Flume original generation），由 Cloudera 公司开发，叫做 Cloudera Flume；后来 Cloudera 把 Flume 贡献给了 Apache，版本改为 FlumeNG（Flume next generation），现在称为 Apache Flume。

注意： 当前大部分高级数据源都没有提供针对 Python API 的访问接口，因此使用 Python API 时，先

在官方网站上确认是否支持。

Flume 是一个分布式的、可靠的、可以用于有效收集、聚合、移动海量日志数据的服务。Flume 以 Agent 为最小的独立运行单位。单个 Agent 由 Source、Sink 和 Channel 三大组件构成，如图 4.39 所示。

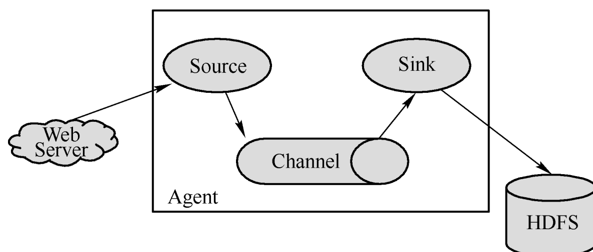


图 4.39　Flume Agent 三大组件图

Flume 提供了大量内置的 Source、Channel 和 Sink 类型。不同类型的 Source、Channel 和 Sink 可以自由组合。组合方式基于用户设置的配置文件。

Agent 的配置信息包括以下几部分：

1）Agent 的 Source 信息，多个 Source 间空格分隔。

2）Agent 的 Channel 信息，多个 Channel 间空格分隔。

3）Agent 的 Sink 信息，多个 Sink 间空格分隔。

4）各个 Source 与 Channel 的关联，多个 Channel 间空格分隔。

5）各个 Sink 与 Channel 的管理，多个 Channel 间空格分隔。

其中，Source 作为数据来源，Channel 作为 Flume 的数据缓存，Sink 是各个数据接收端。指定关联关系后，数据就会从 Source 端开始收集数据，然后发送到关联的 Channel 上进行缓存，最后发送到与 Channel 的 Sink 上。

在基于 Flume 风格的推送数据方式，和定义 Sink 的拉取数据方式的案例实践与分析中，Sink 的 host：port 设置要求不同，可以简单地用以下方法加强理解：

1）使用推送数据方式时，由 Flume 负责推送数据，因此需要知道推送的目的地“host：port”，所以对应的 Sink 设置的“host：port”就是我们集群中的某个 worker 节点地址。

2）使用拉取数据方式时，原理同上，Spark Streaming 应用从 Flume 侧拉取数据，因此需要知道 Flume 侧的“host:port”，所以对应定制的 Sink 设置的“host:port”是在 Agent 运行节点上。运行 Spark Streaming 时，输入该地址，就可以从该地址拉取数据了。

4.3.7　基于 Flume 风格的推送数据案例与解析

基于 Flume 风格的推送数据方式，本质上是 Spark Streaming 在 Flume 中建立一个 Avro agent 的接收器，用于接收流数据，比如企业要处理的海量日志数据，然后在 Flume 收集到数据后推送到 Spark 进行处理。

一般需要满足以下两个要求：

1）当“Flume + Spark Streaming”应用程序启动时，其中一台 Spark Worker 节点必须是

Flume 推送数据的那台机器。

2）同时，Flume 必须可以向这台机器的指定端口推送数据。

一、配置 Flume



基于推送方式的流数据应用，需要为 Flume Agent 配置一个具有如下格式的 Avro sink 配置信息，如下所示：

```
agent. sinks = avroSink
agent. sinks. avroSink. type = avro
agent. sinks. avroSink. channel = memoryChannel
agent. sinks. avroSink. hostname = < chosen machine' s hostname >
agent. sinks. avroSink. port = < chosen port on the machine >
```

其中，agent 是配置的 Flume Agent 的名字，avroSink 为接收数据的 Sink 的名字，memoryChannel 为该 Sink 关联的 Channel 的名字，hostname 和 port 指定 Sink 所在的机器信息。

当前“Flume + Spark Streaming”应用程序需要添加的依赖包：

```
groupId = org. apache. Spark
artifactId = Spark - streaming - flume_2. 10
version = 1. 3. 0
```

可以在 SBT 或 Maven 的构建文件中添加该依赖信息，或者在 IDEA 中将该 jar 包添加到 Libraries 中。

注意：配置信息中的 hostname 必须和集群的资源管理器节点的 hostname 一样，这样在资源调度时，才能匹配名字并在对应的机器节点上启动 Receiver。

部署时，可以将 spark - submit 没有提供的依赖包一起打包到应用程序的 jar 包中，这样在 spark - submit 提交时，可以不需要通过 -- jars 等方式增加外部依赖包。

二、Flume + Spark Streaming 实践案例与解析

（一）Flume Agent 配置案例与解析

配置文件 avro. conf 的案例与解析：

```
## Describe the agent
a1. sources = r1
a1. sinks = k1avroSink
a1. channels = c1 c2
# Describe/configure the source
a1. sources. r1. type = avro
a1. sources. r1. bind = wxx215
a1. sources. r1. port = 4141
# Describe the sink
a1. sinks. k1. type = logger
a1. sinks. avroSink. type = avro
a1. sinks. avroSink. hostname = wxx225
a1. sinks. avroSink. port = 4142
# Use a channel which buffers events in memory
a1. channels. c1. type = memory
a1. channels. c1. capacity = 1000
a1. channels. c1. transactionCapacity = 100
```

```
a1. channels. c2. type = memory
a1. channels. c2. capacity = 1000
a1. channels. c2. transactionCapacity = 100
# Bind the source and sink to the channel
a1. sources. r1. channels = c1  c2
a1. sinks. k1. channel = c1
a1. sinks. avroSink. channel = c2
```

对应的数据流如图 4. 40 所示。

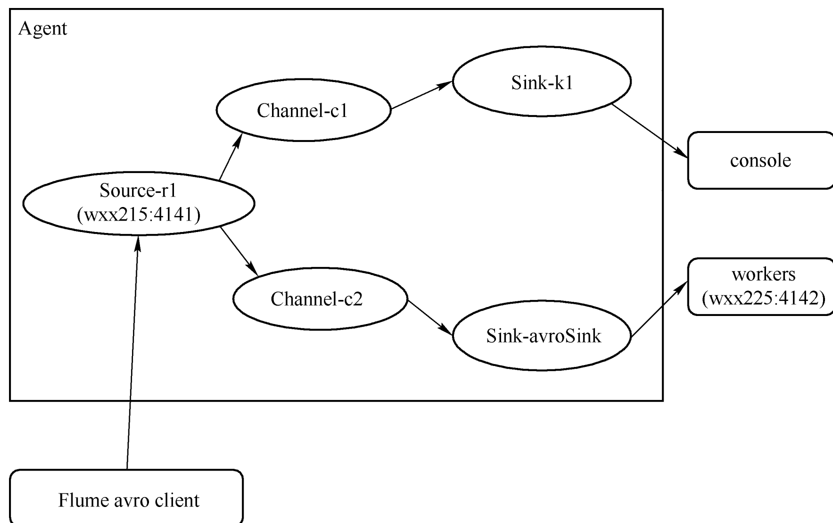


图 4. 40　当前配置对应的数据流图

配置项解析的具体内容参考表 4. 2。

表 4. 2　配置项及其说明

配　置　项	说　　明	备　　注
a1	Agent 的名字	在使用 flume – ng agent 命令启动一个Agent 时，用 – n 选项指定要使用的 Agent 名字
k1avroSink	当前同时设置了两个 Sink	K1：用于界面监控 avroSink：用于 Spark 接收数据
c1 c2	这里配置了两个 Channel	每个 Sink 关联一个 Channel，同时唯一的 source：r1，同时关联到这两个 Channel
a1. sources. r1. bind = wxx215 a1. sources. r1. port = 4141	Source，即设置了数据来源，通过指定的 hostname 与 port 来获取数据	在使用 Flume 提供的 flume – ng avro – client 工具时，向该指定的地址发送文件
a1. sinks. avroSink. type = avro a1. sinks. avroSink. hostname = wxx225 a1. sinks. avroSink. port = 4142	这是 Spark 用来接收数据的配置信息	该接收地址必须在 Spark Streaming 调度的 Workers 节点上

（二）应用程序案例与解析——对应单 Receiver 场景

应用程序代码：

```
package stream
import org. apache. spark. SparkConf
```

```
import org.apache.spark.examples.streaming.StreamingExamples
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming._
import org.apache.spark.streaming.flume._
import stream.util.IntParam
object FlumeEventCount {
  def main(args:Array[String]) {
    if (args.length < 2) {
      System.err.println(
        "Usage:FlumeEventCount <host> <port>")
      System.exit(1)
    }
    //屏蔽过多的日志信息
    import org.apache.log4j.{Level,Logger}
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.streaming").setLevel(Level.WARN)
    //该句默认情况下不起作用
StreamingExamples.setStreamingLogLevels()
    val Array(host,IntParam(port)) = args
    //val batchInterval = Milliseconds(2000)
    val batchInterval = Seconds(10)
    //Create the context and set the batch size
    val sparkConf = new SparkConf().setAppName("FlumeEventCount")
    val ssc = new StreamingContext(sparkConf,batchInterval)
    //Create a flume stream
    val stream = FlumeUtils.createStream(ssc,host,port,StorageLevel.MEMORY_ONLY_SER_2)
    //Print out the count of events received from this server in each batch
    stream.count().map(cnt => "Received " + cnt + " flume events.").print()
ssc.start()
ssc.awaitTermination()
  }
```

（三）开始案例实践与解析

1. 使用 avro.conf 配置属性文件，启动 Flume Agent，命令如下：

```
[harli@wxx215 apache-flume-1.5.2-bin]$./bin/flume-ng agent -c conf -f./harliconf/avro.conf
 -n a1 -Dflume.root.logger=INFO,console
```

其中：

1） -c：指定当前配置文件的目录，设置为 conf。

2） -f：指定当前使用的配置文件。

3） -n：指定配置文件中要启动的 Agent 名字。

当还没有启动 Spark Streaming 应用服务时，Agent 向 avroSink 指定的地址发送信息会失败，错误信息类似于：

```
15/04/15 14:28:40 ERROR flume.SinkRunner:Unable to deliver event. Exception follows.
org.apache.flume.EventDeliveryException:Failed to send events
```

```
…
Caused by: org. apache. flume. FlumeException: NettyAvroRpcClient { host: wxx225, port: 4142 }: RPC connection error
…
Caused by: java. io. IOException: Error connecting to wxx225/192. 168. 70. 225: 4142
…
Caused by: java. net. ConnectException: 拒绝连接
        at sun. nio. ch. SocketChannelImpl. checkConnect( Native Method)
        at sun. nio. ch. SocketChannelImpl. finishConnect( SocketChannelImpl. java: 701)
        at…
…
```

可以先忽略，等 Spark Streaming 应用程序启动后，打开 Receiver 开始接收数据后，Agent 的推送数据就不会出现连接失败的错误了。

2. Avro 客户端模拟

Flume 内部提供了一个 Avro Client 的实现，可以通过 avro – client 向 Agent 提供数据，发送数据的 avro – client 命令，指定 source – r1 的 hostname 和 port：

```
./bin/flume – ng avro – client – – conf avro. conf – H wxx225 – p 4141 – F. . /test. xml – Dflume. root. logger = DEBUG, console( )
```

3. 启动 Spark Streaming 应用

```
./bin/spark – submit   – – master spark: //192. 168. 70. 214: 7077 \
                        – – deploy – mode client \
                        – – driver – memory 1g \
                        – – driver – cores 1 \
                        – – total – executor – cores 3 \
                        – – executor – memory 1g \
                        – – class stream. FlumeEventCount \
                        – – jars. /lib/spark – examples – 1. 3. 0 – hadoop2. 4. 0. jar \
                       . . /applications/testprojectide. jar wxx225 4142
```

注意：这里的 host 和 port，就是 Flume Agent 的 avroSink 要发送到的目的地址，也就是 Worker 节点上，构建 Receiver 所需要的地址。

启动后，打开 Web Interface 界面（http: //wxx225: 4040），其中，wxx225 是启动应用的 Driver Program 所在节点。可以看到 Spark Streaming 应用增加了 Streaming 的信息，如图 4. 41 所示。

这是总体统计信息，包含了以下内容：

1）Receiver 个数，因为当前只设置了一个 avroSink，同时其对应的 port 仅配置了一个，因此这里只有一个 Receiver。

2）Batch interval：对应批数据的时间片大小，这里是代码中设置的 10 秒。

3）Processed batches：这表示到目前为止一共处理了多少批数据。

4）Waiting batches：等待处理的批数据个数。

5）Received records：接收到的记录条数。

6）Processed records：已经处理的记录条数。

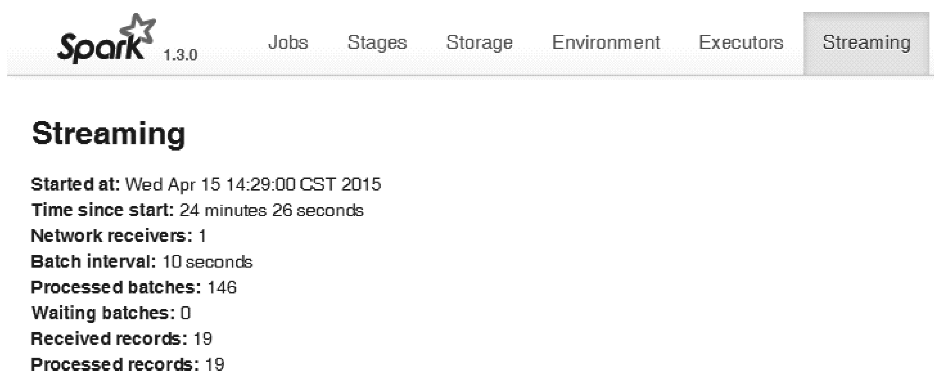


图 4. 41 Spark Streaming 应用的 Streaming 界面

再下面是一些统计信息，如图 4. 42 所示。

Statistics over last 100 processed batches

Receiver Statistics

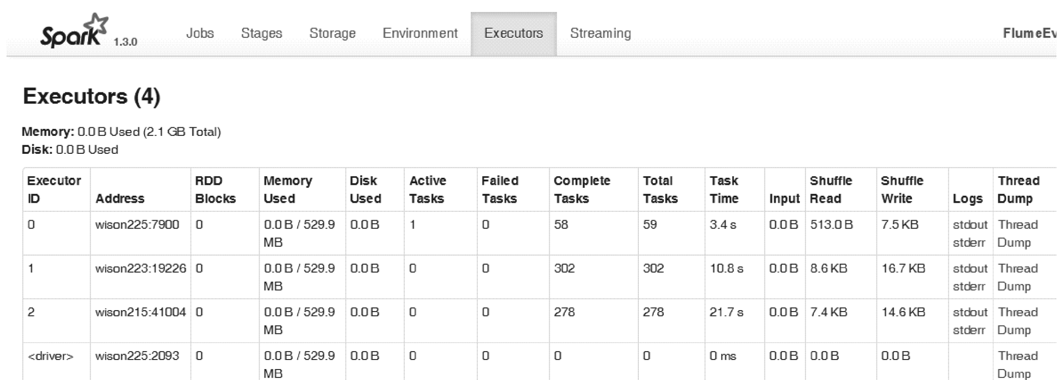
Receiver	Status	Location	Records in last batch [2015/04/15 14:53:26]
FlumeReceiver-0	ACTIVE	wison225	0

Batch Processing Statistics

Metric	Last batch	Minimum	25th percent
Processing Time	65 ms	65 ms	89 ms
Scheduling Delay	0 ms	0 ms	0 ms
Total Delay	65 ms	65 ms	90 ms

图 4. 42 Spark Streaming 的 Streaming 界面统计信息

这是 Executor 页面的信息，如图 4. 43 所示。


Spark 1.3.0 Jobs Stages Storage Environment Executors Streaming FlumeEv

Executors (4)

Memory: 0.0 B Used (2.1 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
0	wison225:7900	0	0.0 B / 529.9 MB	0.0 B	1	0	58	59	3.4 s	0.0 B	513.0 B	7.5 KB	stdout stderr	Thread Dump
1	wison223:19226	0	0.0 B / 529.9 MB	0.0 B	0	0	302	302	10.8 s	0.0 B	8.6 KB	16.7 KB	stdout stderr	Thread Dump
2	wison215:41004	0	0.0 B / 529.9 MB	0.0 B	0	0	278	278	21.7 s	0.0 B	7.4 KB	14.6 KB	stdout stderr	Thread Dump
<driver>	wison225:2093	0	0.0 B / 529.9 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

图 4. 43 Spark Streaming 的 Executor 页面

可以看到，当前使用了 3 个 Executor。其中 Active Tasks 列，对应了 Receiver 所在的节点。也是我们在配置文件中指定的数据 avroSink 的 hostname 对应的节点。

4. 启动 Flume Avro Client 发送数据

这时，对应的 Agent 界面（即 k1 这个 Sink）输出内容如下：

```
15/04/15 15:02:14 INFO source. AvroSource:Avro source r1 started.
15/04/15 15:02:32 INFO ipc. NettyServer:[id:0x32506a15,/192.168.70.225:45288 => /192.168.70.215:4141] OPEN
15/04/15 15:02:32 INFO ipc. NettyServer:[id:0x32506a15,/192.168.70.225:45288 => /192.168.70.215:4141] BOUND:/192.168.70.215:4141
15/04/15 15:02:32 INFO ipc. NettyServer:[id:0x32506a15,/192.168.70.225:45288 => /192.168.70.215:4141] CONNECTED:/192.168.70.225:45288
15/04/15 15:02:33 INFO ipc. NettyServer:[id:0x32506a15,/192.168.70.225:45288:> /192.168.70.215:4141] DISCONNECTED
15/04/15 15:02:33 INFO ipc. NettyServer:[id:0x32506a15,/192.168.70.225:45288:> /192.168.70.215:4141] UNBOUND
15/04/15 15:02:33 INFO ipc. NettyServer:[id:0x32506a15,/192.168.70.225:45288:> /192.168.70.215:4141] CLOSED
15/04/15 15:02:33 INFO ipc. NettyServer:Connection to /192.168.70.225:45288 disconnected.
15/04/15 15:02:36 INFO sink. LoggerSink:Event:{ headers:{} body:5B 32 30 31 35 2D 30 34 2D 31 34 20 31 32 3A 30 [2015-04-14 12:0 }
15/04/15 15:02:36 INFO sink. LoggerSink:Event:{ headers:{} body:5B 32 30 31 35 2D 30 34 2D 31 34 20 31 32 3A 30 [2015-04-14 12:0 }
15/04/15 15:02:36 INFO sink. LoggerSink:Event:{ headers:{} body:5B 32 30 31 35 2D 30 34 2D 31 34 20 31 32 3A 30 [2015-04-14 12:0 }
15/04/15 15:02:36 INFO sink. LoggerSink:Event:{ headers:{} body:5B 32 30 31 35 2D 30 34 2D 31 34 20 31 32 3A 30 [2015-04-14 12:0 }
15/04/15 15:02:36 INFO sink. LoggerSink:Event:{ headers:{} body:5B 32 30 31 35 2D 30 34 2D 31 34 20 31 32 3A 30 [2015-04-14 12:0 }
15/04/15 15:02:36 INFO sink. LoggerSink:Event:{ headers:{} body:5B 32 30 31 35 2D 30 34 2D 31 34 20 31 32 3A 30 [2015-04-14 12:0 }
……
```

而在 Spark Streaming 的 Driver 节点上，也就是当前启动应用的 wxx225 节点上，也会有对应的界面输出信息，这是对 avroSink 收集数据的处理结果。

为了测试 Flume 与 Spark 间的收发数据，先停止应用程序，停止期间使用 Flume Avro Client 工具发送两次文件（一个文件对应有 19 条 Event 记录），然后，开始启动应用程序，完全启动后，再使用 Flume avroclient 工具发送一次文件，这时候对应的界面信息如下：

```
[harli@wxx214 spark-1.3.0-bin-hadoop2.4]$ ./bin/spark-submit    --master
spark://192.168.70.214:7077                    --deploy-mode client
 --driver-memory 1g                            --driver-cores 1
 --total-executor-cores 3                      --executor-memory 1g
 --class stream.FlumeEventCount
 --jars ./lib/spark-examples-1.3.0-hadoop2.4.0.jar         ../applications/testprojectide.jar
wxx225 4142
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
15/04/15 14:11:05 WARN util. NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/04/15 14:11:06 INFO slf4j. Slf4jLogger:Slf4jLogger started
```

```
15/04/15 14:11:06 INFORemoting:Starting remoting
15/04/15 14:11:07 INFORemoting:Remoting started; listening on addresses:[akka.tcp://sparkDriver@wxx214:53437]
15/04/15 14:11:07 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/15 14:11:07 INFO server.AbstractConnector:Started SocketConnector@0.0.0.0:59253
15/04/15 14:11:07 INFO server.Server:jetty-8.y.z-SNAPSHOT
15/04/15 14:11:07 INFO server.AbstractConnector:Started SelectChannelConnector@0.0.0.0:4040
-------------------------------------------
Time:1429078280000 ms
-------------------------------------------
Received 0 flume events.
-------------------------------------------
Time:1429078290000 ms
-------------------------------------------
Received 0 flume events.
-------------------------------------------
Time:1429078300000 ms
-------------------------------------------
Received38 flume events.
-------------------------------------------
Time:1429078310000 ms
-------------------------------------------
Received19 flume events.
-------------------------------------------
Time:1429078320000 ms
-------------------------------------------
Received 0 flume events.
```



即，Flume 会在 Channel 中缓存数据直到 Sink 成功将数据推送到 Spark。

在 Spark 1.3 版本中已经将 Flume 访问放到 External 部分，在实际执行时，不用将依赖的 Flume 打包进来就可以直接提交应用。

4.3.8 定制 FlumeSink 的拉取数据案例与解析

除了基于 Flume 风格的推送数据的方式外，还可以使用定制的 Sink，基于拉取数据的方式整合 Flume 与 Spark Streaming。使用拉取数据的方式有以下两个优点：

1）Flume 将数据推送到定制的 Sink，并且这些数据会被缓存起来。

2）Spark Streaming 使用一个可靠的 Flume Receiver 和事务处理从 Sink 中拉取数据。事务仅仅在数据接收并在 Spark Streaming 中备份后才算成功。

拉取数据方式相比第一种方法，可以确保强可靠性以及增加了对容错方面的保证。

拉取数据方式的一般性的要求：需要选择一台机器，在一个 Flume Agent 上运行定制的 Sink，Spark 机器中的机器节点必须可以访问运行定制 Sink 的这台机器。

（一）配置 Flume

1. 在负责运行定制 Sink 的节点上，将定制的 Sink 对应的 jar 包添加到 Flume 的 lib 路径下。

定制的 Sink 对应的 jar 信息如下：

```
groupId = org. apache. spark
artifactId = spark - streaming - flume - sink_2. 10
version = 1. 3. 0
```

可以到 Maven 仓库手动下载该 jar 包。

2. 在该机器上，为 Flume Agent 设置定制的 Sink 相关的配置信息，信息内容如下所示：

```
agent. sinks = spark
agent. sinks. spark. type = org. apache. spark. streaming. flume. sink. SparkSink
agent. sinks. spark. hostname = <hostname of the local machine>
agent. sinks. spark. port = <port to listen on for connection from Spark>
agent. sinks. spark. channel = memoryChannel
```

其中：

1） agent 对应 Flume Agent 的名字。

2） Spark 对应定制的 Sink 的名字。

3） org. apache. spark. streaming. flume. sink. SparkSink 对应的是定制 Sink 的类名，由于不是 Flume 内置的 Sink，因此需要使用全路径。

4） memoryChannel 对应定制的 Sink 所关联的 Channel 名字。

5） hostname 与 port 是接收数据的机器信息。

（二）Flume +Spark Streaming 实践案例与解析

同样，整合应用程序也需要添加 sparkSpark - streaming - flume_2. 10 依赖包。对应的部署方式也一样。

（三）Flume Agent 配置案例与解析

复制基于 Flume 风格的推送数据方式中的配置文件 avro. conf 为 spark. conf。然后，将其中的 Sink 的名字 avroSink 修改为 sparkSink，同时，修改该 Sink 的 type 信息，由 avro 修改为定制的 Sink，即 org. apache. spark. streaming. flume. sink. SparkSink。

对应的配置文件内容如下：

```
[harli@ wxx215 apache - flume - 1. 5. 2 - bin] $  vim harliconf/spark. conf
a1. sources = r1
a1. sinks = k1 sparkSink
a1. channels = c1 c2
# Describe/configure the source
a1. sources. r1. type = avro
a1. sources. r1. bind = wxx215
a1. sources. r1. port = 4141
# Describe the sink
a1. sinks. k1. type = logger
a1. sinks. sparkSink. type = org. apache. spark. streaming. flume. sink. SparkSink
a1. sinks. sparkSink. hostname = wxx215
a1. sinks. sparkSink. port = 4142
# Use a channel which buffers events in memory
a1. channels. c1. type = memory
```

```
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
a1.channels.c2.type = memory
a1.channels.c2.capacity = 1000
a1.channels.c2.transactionCapacity = 100
# Bind the source and sink to the channel
a1.sources.r1.channels = c1 c2
a1.sinks.k1.channel = c1
a1.sinks.sparkSink.channel = c2
~
```

配置文件中，重点关注 sparkSink 的属性配置，这里配置为 Agent 所在的 hostname，Spark Streaming 应用启动后，通过该 hostname 和 port 信息，拉取信息。

当前配置文件中的配置对应的数据流如图 4.44 所示。

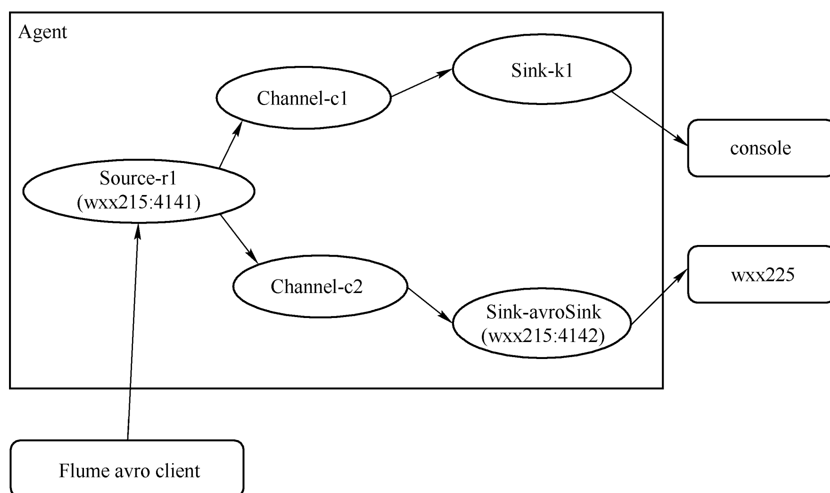


图 4.44 当前配置对应的数据流图

用新配置的属性文件 spark.conf，启动 Flume Agent：

```
[harli@wxx215 apache-flume-1.5.2-bin]$ ./bin/flume-ng agent -c conf -f ./harliconf/spark.conf -n a1 -Dflume.root.logger=INFO,console
……

15/04/15 16:34:03 ERROR node.PollingPropertiesFileConfigurationProvider: Failed to load configuration data. Exception follows.
org.apache.flume.FlumeException: Unable to load sink type: org.apache.spark.streaming.flume.sink.SparkSink, class: org.apache.spark.streaming.flume.sink.SparkSink
  ……
Caused by: java.lang.ClassNotFoundException: org.apache.spark.streaming.flume.sink.SparkSink
  ……
```

由于 sparkSink 上定制的 Sink，不是 Flume 内置的，使用时，需要先将定制的 Sink 对应的 jar 包（spark-streaming-flume-sink_2.10-1.3.0.jar）添加到启动 Agent 时的 CLASS-

PATH 路径下，这里复制到 $ FLUME_HOME/lib 路径下：

```
[harli@ wxx215 lib] $ cp /usr/harli/spark - streaming - flume - sink_2. 10 - 1. 3. 0. jar  ./
[harli @ wxx215 apache - flume - 1. 5. 2 - bin] $ ls ./lib/spark - streaming - flume - sink_2. 10
- 1. 3. 0. jar
./lib/spark - streaming - flume - sink_2. 10 - 1. 3. 0. jar
[harli@ wxx215 apache - flume - 1. 5. 2 - bin] $
```

重新启动 Agent，启动过程会出现一些类找不到的错误提示，这是因为定制的 Sink 所在的 jar 包依赖了 Scala 的 jar 包和 Spark 的 jar 包。Flume 中用到了 Hadoop 的类库，启动 Flume Agent 时会自动去识别环境变量 $ HADOOP_HOME，然后添加 Hadoop 的类库到 Flume 运行的 CLASSPATH 下，可以参看启动 Agent 时，界面输出信息中的 CLASSPATH 内容，是包含 hadoop lib 下的 jar 包的，也就是说，如果环境中没有配置 Hadoop 的环境变量 $ HADOOP_HOME，启动 Agent 时也会报类找不到的错误。而对应的 Scala 和 Spark 类库，则是因为引入定制的 Sink 才需要的，因此默认情况下是不会自动识别 $ SCALA_HOME 和 $ Spark_HOME 这两个环境变量，然后自动添加所需 jar 包到 Flume 的 CLASSPATH 路径下的。

这本机上测试时，错误信息如下：

1. 找不到 Scala 类的错误信息

```
15/04/15 06:17:33 ERROR node. PollingPropertiesFileConfigurationProvider:Failed to start agent be-
cause dependencies were not found in classpath.  Error follows.
java. lang. NoClassDefFoundError:scala/Function1
……
    at java. lang. Thread. run(Thread. java:745)
Caused by:java. lang. ClassNotFoundException:scala. Function1
……
```

2. 找不到 Spark 类的错误信息

```
15/04/15 06:14:40 ERROR lifecycle. LifecycleSupervisor: Unable to start SinkRunner: { policy:
org. apache. flume. sink. DefaultSinkProcessor@ 3e7dad3b counterGroup: { name:null counters:{ } } }
 - Exception follows.
java. lang. NoClassDefFoundError:org/spark - project/guava/util/concurrent/ThreadFactoryBuilder
……
    at java. lang. Thread. run(Thread. java:745)
Caused by: java. lang. ClassNotFoundException: org. spark -
project. guava. util. concurrent. ThreadFactoryBuilder
……
```

解决方法，是将 Scala 和 Spark 的类库放入 Flume 的 CLASSPATH（对应用 Java 命令执行时的 -cp 选项）下，具体步骤如下：

1）编辑 Flume 的环境变量配置文件 conf/flume - env. sh。

2）将 Scala 和 Spark 的 lib 类库添加到环境变量，如下所示：

```
#FLUME_CLASSPATH = ""
FLUME_CLASSPATH = " $ SCALA_HOME/lib/ * : $ Spark_HOME/lib/ * "
```

在启动过程中，还可能会出现以下错误，如 Flume 的方法找不到的错误信息：

```
15/04/15 06:46:06 ERROR node. PollingPropertiesFileConfigurationProvider:Unhandled error
java. lang. NoSuchMethodError: org. apache. flume. Context. getSubProperties ( Ljava/lang/String; ) Lcom/
google/common/collect/ImmutableMap;
……
at java. lang. Thread. run( Thread. java:745)
```

在启动时，将 -c 选项设置为 Flume 的 conf 即可。

小技巧：在 vim 中替换全局的字符串，可以使用“s/avroSink/sparkSink/g”这种方式，会将文件文件中第一个字符“avroSink”全部替换为“sparkSink”；输入/string 可以搜索 string 字符串，输入 n 可以查找下一个。具体请参考 vim 使用手册。

修改 CLASSPATH 成功，指定 sparkSink 的“host：port”，再次启动 Spark Streaming 应用，比如：

```
./bin/spark-submit   --master spark://192.168.70.214:7077 \
                      --deploy-mode client \
                      --driver-memory 1g \
                      --driver-cores 1 \
                      --total-executor-cores 2 \
                      --executor-memory 1g \
                      --class stream.FlumePollingEventCount \
                      --jars ./lib/spark-examples-1.3.0-hadoop2.4.0.jar  \
                     ../applications/testprojectide.jar wison225 4142
```

再次向 source-r1 指定的“host: port”发送数据：

```
[harli@wison225 apache-flume-1.5.2-bin]$ ./bin/flume-ngavro-client -c . -H wison215
-p 4141 -F ../test/log.txt
```

查看 Driver 的 Web Interface 界面（http://wxx215:4040）的 Streaming 页面，如图 4.45 所示。

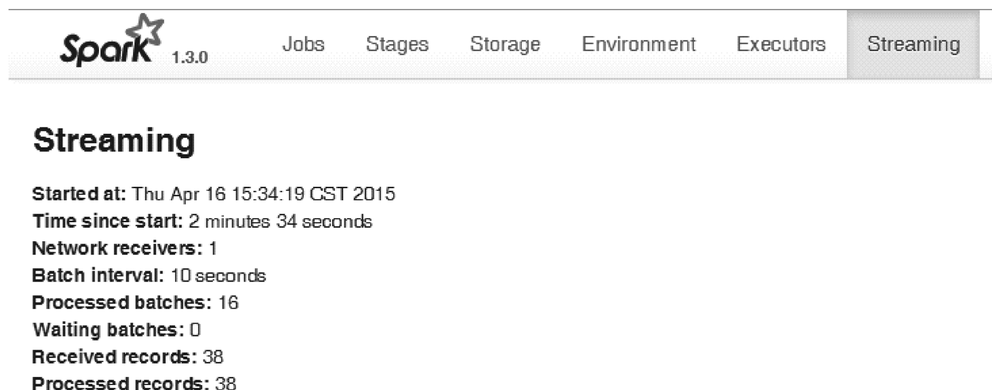


图 4.45 Spark Streaming 的 Streaming 页面

以上是定制 Sink 在单 Receiver 的使用场景，下面给出定制 Sink 在两种场景下，Flume 的配置及相应代码的案例实践与解析，注意拉取方式下定制 Sink 的 host：port 的设置，参考单 Receiver 案例即可。这里采用单机方式进行案例实践，此时，需要注意的是，要通过 port

来区分不同的 Source、Channel、Sink。

（四）共享 Channel 场景下的多并行度的应用程序案例与解析

在实际企业级应用场景下，为了提高接收数据的并行度，需要相应的增加推送数据的 Sink 配置。需要注意的是，Flume 在推送数据时，是在 Sink 推送后，就清除掉 Channel 里的数据，因此，如果两个 Sink 关联到同一个 Channel 的话，实际推送时，就相当于队列分发模式列。下面如测试这种情况。

这里在自己的集群（仅 Cluster01 节点的分布式集群）上测试，配置文件 c2Spark. conf 如下：

```
# Describe/configure the source
a1. sources. r1. type = avro
a1. sources. r1. channels = c1
a1. sources. r1. bind = cluster01
a1. sources. r1. port = 4141
# Describe the sink
a1. sinks. k1. type = logger
a1. sinks. sparkSink. type = org. apache. spark. streaming. flume. sink. SparkSink
a1. sinks. sparkSink. hostname = cluster01
a1. sinks. sparkSink. port = 4142
a1. sinks. sparkSink2. type = org. apache. spark. streaming. flume. sink. SparkSink
a1. sinks. sparkSink2. hostname = cluster01
a1. sinks. sparkSink2. port = 4143
# Use a channel which buffers events in memory
a1. channels. c1. type = memory
a1. channels. c1. capacity = 1000
a1. channels. c1. transactionCapacity = 100
a1. channels. c2. type = memory
a1. channels. c2. capacity = 1000
a1. channels. c2. transactionCapacity = 100
# Bind the source and sink to the channel
a1. sources. r1. channels = c1 c2
a1. sinks. k1. channel = c1
a1. sinks. sparkSink. channel = c2
a1. sinks. sparkSink2. channel = c2
```

注意：这里 sparkSink2 的配置和 sparkSink 基本一样，和 sparkSink 使用了相同的 channel - c2，唯一不同的地方在于 port 的设置（这是因为当前在一台机器上）。

启动 Flume Agent：

```
[harli@ cluster01 flume] $   ./bin/flume - ng agent  - c conf  - f ./harliconf/errorspark. conf  - n a1
 - Dflume. root. logger = INFO,console
Info:Sourcing environment configuration script /home/harli/cluster/flume/conf/flume - env. sh
Info:IncludingHadoop libraries found via (/home/harli/cluster/hadoop/bin/hadoop) for HDFS access
Info: Excluding /home/harli/cluster _ 13/hadoop/share/hadoop/common/lib/slf4j - api - 1. 7. 5. jar
from classpath
Info:Excluding /home/harli/cluster_13/hadoop/share/hadoop/common/lib/slf4j - log4j12 - 1. 7. 5. jar
from classpath
```

启动过程中，虽然配置文件中两个 Sink 关联到了同一个 Channel，但启动不会报错或警告。

打开两个终端，启动两个 Spark Streaming 应用：

```
[harli@cluster01 spark]$ ./bin/spark-submit    --master spark://cluster01:7077
 --deploy-mode client                                    driver-memory 1g
 --driver-cores 1                                 --total-executor-cores 2
 --executor-memory 1g                 --class stream.FlumePollingEventCount
 --jars ./lib/spark-examples-1.3.0-hadoop2.4.0.jar
                          ../applications/testprojectide.jar cluster01 4142
Spark assembly has been built with Hive,includingDatanucleus jars on classpath

[harli@cluster01 spark]$ ./bin/spark-submit    --master spark://cluster01:7077
 --deploy-mode client                                    driver-memory 1g
 --driver-cores 1                                 --total-executor-cores 2
 --executor-memory 1g                       --class stream.FlumePollingEventCount
 --jars ./lib/spark-examples-1.3.0-hadoop2.4.0.jar
                          ../applications/testprojectide.jar cluster01 4143
Spark assembly has been built with Hive,includingDatanucleus jars on classpath
```

查看 Master 的 Web Interface 界面（http://cluster01:8080/），如图 4.46 所示。可以看到同时运行了两个 Spark streaming 应用程序。

cluster01:8080

Spark 1.3.0 **Spark Master at spark://cluster01:7077**

URL: spark://cluster01:7077
REST URL: spark://cluster01:6066 *(cluster mode)*
Workers: 1
Cores: 4 Total, 4 Used
Memory: 2.7 GB Total, 2.0 GB Used
Applications: 2 Running, 4 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Core
worker-20150415043845-cluster01-48448	cluster01:48448	ALIVE	4 (4

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted T
app-20150415071733-0005	FlumePollingEventCount	2	1024.0 MB	2015/04/15 (
app-20150415071720-0004	FlumePollingEventCount	2	1024.0 MB	2015/04/15 (

图 4.46 Spark Streaming 的应用信息

各自单击应用程序的名字，进入 Spark Streaming 应用的 Driver 的 Web Interface 界面，打开 Streaming 页面，如图 4.47 所示。持续一段时间后，Streaming 页面信息的变更为如图 4.48 所示。

两个 Spark Streaming 应用程序都收到了从 Channel-c2 上拉取过来的数据。

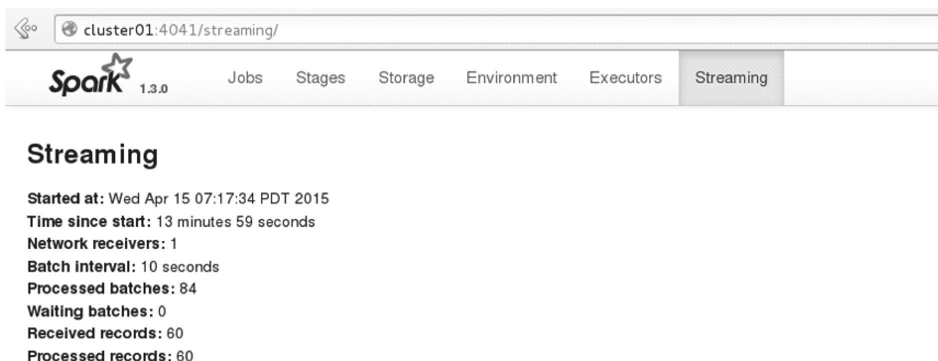


图 4.47 Spark Streaming 的应用的 Streaming 页面信息

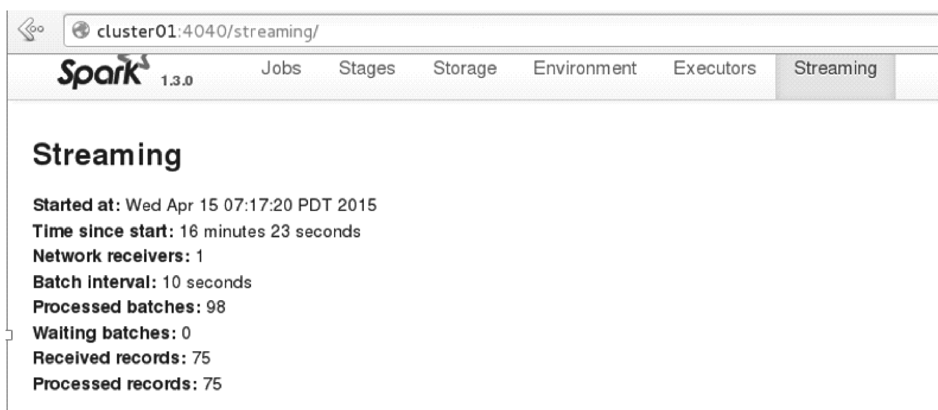


图 4.48 一段时间后 Streaming 页面信息的变更

（五）Sink 与 Channel 一对一场景下的多并行度的应用程序案例与解析

一般情况下，为了提高集群的资源使用率。应用程序的性能等，都需要同时启动多个 Receivers，增加数据接收的并行度，下面在单 Receiver 基础上给出新增一个 Receiver 的案例及分析。

还可以更进一步，即在一个 Agent 上添加一个新的 Resource，从整条数据流上增加并行度，在方法上一样的。这里为了简化，只简单添加新的 Sink 和 Channel。

当然，企业级别在实际应用时，可以为各个数据源都配置一个 Agent 来收集数据，只要将配置文件复制过去，然后启动 Agent 即可。

Flume 的配置 multispark. conf：

```
a1. sources = r1
a1. sinks = k1 sparkSink sparkSkink2
a1. channels = c1 c2 c3
# Describe/configure the source
a1. sources. r1. type = avro
a1. sources. r1. bind = cluster01
a1. sources. r1. port = 4141
# Describe the sink
a1. sinks. k1. type = logger
a1. sinks. sparkSink. type = org. apache. spark. streaming. flume. sink. SparkSink
```

```
a1.sinks.sparkSink.hostname = cluster01
a1.sinks.sparkSink.port = 4142
a1.sinks.sparkSkink2.type = org.apache.spark.streaming.flume.sink.SparkSink
a1.sinks.sparkSkink2.hostname = cluster01
a1.sinks.sparkSkink2.port = 4143
# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
a1.channels.c2.type = memory
a1.channels.c2.capacity = 1000
a1.channels.c2.transactionCapacity = 100
a1.channels.c3.type = memory
a1.channels.c3.capacity = 1000
a1.channels.c3.transactionCapacity = 100
# Bind the source and sink to the channel
a1.sources.r1.channels = c1 c2
a1.sinks.k1.channel = c1
a1.sinks.sparkSink.channel = c2
a1.sinks.sparkSink2.channel = c3
~
```



配置文件中，增加了一个新的 Sink－sparkSink2，和新的 channel－c3，并将这两者关联起来。

应用程序代码：

```
package stream
import org.apache.spark.SparkConf
import org.apache.spark.examples.streaming.StreamingExamples
import org.apache.spark.streaming._
import org.apache.spark.streaming.flume._
import stream.util.IntParam
object MultiFlumePollingEventCount {
  def main(args:Array[String]) {
    if (args.length < 2) {
      System.err.println(
        "Usage:FlumePollingEventCount <host:port> <host:port>")
      System.exit(1)
    }
    //屏蔽过多的日志信息
    import org.apache.log4j.{Level,Logger}
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.sql").setLevel(Level.WARN)
    Logger.getLogger("org.apache.spark.streaming").setLevel(Level.WARN)
    //该句默认情况下不起作用
StreamingExamples.setStreamingLogLevels()
    val batchInterval = Seconds(10)
    //以指定的批处理时间间隔创建 StreamingContext 实例
    val sparkConf = new SparkConf().setAppName("FlumePollingEventCount")
```

```
    val ssc = new StreamingContext(sparkConf,batchInterval)
    //这里根据输入的 host:port 数组,分别构建一个 flume 流,
    //从运行中 Flume Agent 上的 SparkSink 上拉取数据
    //并且将这几个 flume 流合并为一个流进行处理。
    valflumeStreams = args.map { arg =>
      val Array(host,IntParam(port)) = arg.split(":")
FlumeUtils.createPollingStream(ssc,host,port)
    }
    val unifiedStream = ssc.union(flumeStreams)
    //打印接收到的批数据的 event 数,也就是记录条数
    unifiedStream.count().map(cnt => "Received " + cnt + " flume events.").print()
    //打印的是类信息 org.apache.spark.streaming.flume.SparkFlumeEvent@38f8e8f8
    //应该针对该类型 SparkFlumeEvent,进行具体处理。
    //unifiedStream.print()

ssc.start()
ssc.awaitTermination()
  }
}
```

启动应用命令如下：

```
[harli@cluster01 spark]$ ./bin/spark-submit    --master spark://cluster01:7077
 --deploy-mode client                          --driver-memory 1g
 --driver-cores 1                         --total-executor-cores 3
 --executor-memory 1g                     --class stream.MultiFlumePollingEventCount
 --jars ./lib/spark-examples-1.3.0-hadoop2.4.0.jar                      ../applications/
testprojectide.jar cluster01:4142 cluster01:4143
```

参数 cluster01:4142 cluster01:4143 对应两个 SparkSink 设置的 hostname 和 port 信息。在刚开始启动时，会多次尝试连接，这时候连接失败的信息可以忽略。

然后启动 Flume avro-client 客户端，发送数据：

```
[harli@cluster01 flume]$ ./bin/flume-ngavro-client -c . -H cluster01 -p 4141 -F ../
test/log.txt
```

打开查看 Driver 的 Web Interface 界面（http://clulster01:4040），查看 Streaming 页面信息，如图 4.49 所示。

cluster01:4040/streaming/

Spark 1.3.0 | Jobs | Stages | Storage | Environment | Executors | Streaming

Streaming

Started at: Wed Apr 15 07:59:26 PDT 2015
Time since start: 7 minutes 23 seconds
Network receivers: 2
Batch interval: 10 seconds
Processed batches: 44
Waiting batches: 0
Received records: 63
Processed records: 63

图 4.49　Spark Streaming 应用的 Streaming 页面信息

可以看到已经接收并处理列 63 条记录信息。

在多次快速启动 Flume avro - client 客户端，发送数据后，发现应用程序接收不到数据，这时，查看对应的 Agent 的输出日志，会发现如下报错信息：

```
2015 - 04 - 15 08 : 17 : 37, 353 ( Spark Sink Processor Thread - 9 ) [ WARN - org. apache. spark. streaming. flume. sink. Logging $ class. logWarning ( Logging. scala : 80 ) ] Error while processing transaction.
org. apache. flume. ChannelException:Take list for MemoryTransaction, capacity 100 full, consider committing more frequently, increasing capacity, or increasing thread count
……
2015 - 04 - 15 08 : 17 : 37, 355 ( Spark Sink Processor Thread - 9 ) [ WARN - org. apache. spark. streaming. flume. sink. Logging $ class. logWarning ( Logging. scala : 59 ) ] Spark was unable to successfully process the events. Transaction is being rolled back.
```

从界面日志信息中可以看出，由于当前 Channel 设置的 MemoryTransaction 太小，空间已满，导致无法接收数据。在企业级实际应用中，需要根据实际情况设置足够的 Channel 空间（这里是使用内存进行缓存，还可以使用其他方式，具体参考 Flume 的官方文档说明）。

需要注意的是，在出现以上错误之后，Spark Streaming 就不能继续接收数据了，本机测试时，即使重启 Spark Streaming 应用，也不能继续使用。对应的 Agent 的 Sink - k1，即 console 输出部分，还能正常运行。

Section 4.4 性能调优

从更高层面上来讲，性能调优需要考虑以下两个方面：

1）应该尽可能利用集群资源来减少每个批处理的时间。

2）批处理的数据大小应该尽量合理，保证接收到的数据能及时处理掉。

4.4.1 减少批处理的时间

减少批处理时间，在调优指南上也对细节做了一定的分析，这里着重分析比较重要的几个方面。

一、在数据接收上的并行度

1. Input DStream 的并行度

通过网络从各种数据源（如 Kafka，Flume，Socket 等）接收数据时，会要求将数据反序列化并存储到 Spark。如果数据接收这块变成系统瓶颈的话，就应该考虑提高数据接收的并行度了。

注意：每个 Input DStream 创建一个 Receiver（运行在每个 Worker 节点上），因此只接收一个数据流，因此可以通过构建多个 Input DStreams，并且进行配置，让它们从来自数据源的流数据的不同分区接收数据，相应地，数据源提供的分区个数也就成了 Input DStream 并行度的最大值了。

最后可以将创建的多个 DStream 合并为单个 DStream 进行处理。

2. 任务的并行度——对应 RDD 的分区数

与并行度相关的另一个需要考虑的配置参数是 spark. streaming. blockInterval，这是 Receiver 的块时间间隔。对大部分的 Receivers，接收到的数据在存储到 Spark 内存之前，会先合并到 blocks 中。而这个块的个数，就对应了批数据，也就是 RDD 的分区数，也就是 RDD 的并行任务（task）数了。

因此，tasks 的并行度大致等于“批数据的时间片/接收块的时间间隔”。比如，块间隔时间为 200 毫秒，时间片为 2 秒时，对应的 tasks 就是 2000 毫秒/200 毫秒，也就是并行的 tasks 个数为 10。如果这里并行的 tasks 数远小于集群可用的内核数，则效率较低。因此，在给定的批数据时间片前提下，需要修改块的时间间隔，也就是 spark. streaming. blockInterval，来提高 tasks 的并行度。

一般建议将块的时间的最小值设置为 50 毫秒，如果再低的话，task 启动的开销就会增加。

3. 显示修改并行度

另一个可选的方法是，从多输入流 Receivers 接收数据时，显示地调用重分区的方法（inputStream. repartition 方法）。这样可以在进一步处理数据之前，先把在指定数量的节点上接收到的数据分发到集群上。

二、在数据处理上的并行度

如果计算过程中任何一个 Stage 的任务并行度不够高的话，可能会导致集群资源没有被充分地利用起来。

比如，针对 Key – Value 类型的 DStream 的一些聚合操作，如 reduceByKey 和 reduceByKeyAndWindow 等（与 RDD 类似，对应地在隐式转化的 PairDStreamFunctions 类中），其分区器使用的是默认的分区器，默认分区器的构建可以参考章节 2. 2. 9 分区数设置的案例与源码解析部分，可以通过配置 spark. default. parallelism 属性，来提高分区器的分区个数，当然，也可以通过指定 API 中的并行度参数来显示设置。

三、Data Serialization

可以通过优化序列化的格式来减少数据序列化的开销。在 Spark Streaming 中，有两类数据会被序列化。

1. 输入数据

默认情况下，通过 Receivers 接收的输入数据会被存储在 Executor 的内存中，默认的存储级别为 StorageLevel. MEMORY_AND_DISK_SER_2。这是因为，将数据序列化（_SER）成 bytes 可以减少 GC 的开销，而数据的备份（_2）是为了针对 Executor 故障的容错性。

序列化数据明显是有一定的开销的，首先 Receiver 必须将接收到的数据反序列化，然后再使用 Spark 指定的序列化格式将数据序列化。

2. 在 Spark Streaming 操作中的 RDD 持久化

流计算过程中产生的 RDD 可能会被持久化到内存中。比如，窗口类的操作为了重复处理数据，会将数据持久化到内存中。这和 Spark 内核中 RDD 默认的持久化是不一样的，RDD 默认是使用 StorageLevel. MEMORY_ONLY 进行持久化。

另外，在窗口类的操作过程中，是默认进行持久化的，而 RDD 的持久化是需要人为触发的。

这两种情况下的持久化一般都应该使用 Kryo 序列化，这样可以减少 CPU 和内存的开销。

比较特殊的情况下，可以将数据以反序列化对象进行持久化，只要不会引起高昂的 GC 开销即可。比如，如果批间隔只设置成几秒（对应数据量比较小），就可以通过显示地设定存储级别，去除数据持久化中的序列化。以减少由于序列化引起的 CPU 开销，进而提高性能。

四、Task Launching Overheads

如果每秒钟启动的 task 数过高（比如，每秒启动 50 次或更多时），相应地，向 Slaves 发送 tasks 的开销就比较大了，这会导致很难实现亚秒级的延迟。可以通过以下修改来减低这种开销：

1. task 的序列化：使用 Kryo 序列化机制来序列化 task，来减少 task 的大小，因此也减少了向 Slaves 发送的时间了。

2. 执行模式（Execution Mode）：task 在 Standalone 模式或 coarse－grained Mesos 模式下的启动时间，比在 fine－grained Mesos 模式下要少很多。具体可以从官方网站上，在 Mesos 上运行的指南中获取更详细的信息。

这些修改，可以将批处理时间减少到几百毫秒，进而达到亚秒级的处理。

4.4.2 设置正确的批间隔

想要 Spark Streaming 应用程序能稳定地在集群中运行，系统必须能够尽可能快地处理接收到的数据。也就是说，一旦批数据生成就应该尽快处理掉。

我们可以从 Streaming 的 Web Interface 监控界面上查看相关信息，针对 Spark Streaming 应用程序，Spark 自带的 Driver 的 Web Interface 界面上会相应地增加一个 Streaming 相关信息的 tab 页面，其中包含了两个重要的性能度量指标：

1）处理时间（Processing Time）：即批数据的处理时间。

2）调度延迟（Scheduling Delay）：即每个批数据在队列中等待前一个批数据处理完成所等待的时间。

批处理时间应该小于批数据的时间片，这样才不会出现大的调度延迟，可以避免导致越来越多的批数据等待处理。

下面是 Streaming 监控界面的截图，如图 4.50 所示。

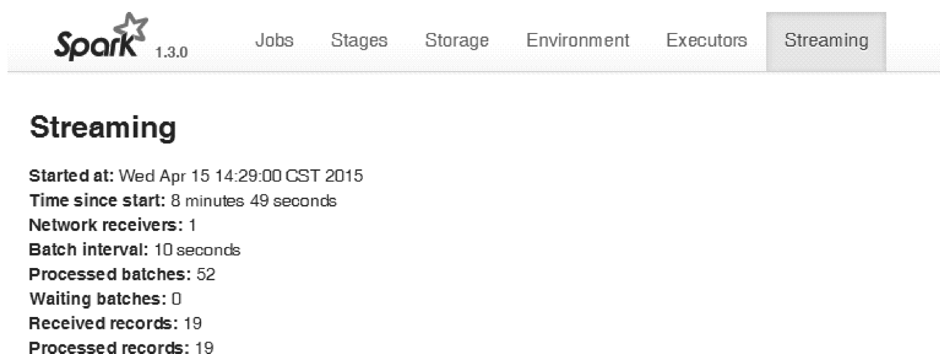


图 4.50　Spark Streaming 应用 Streaming 监控界面

对应的 Spark Streaming 应用的批处理统计信息，如图 4. 50 所示。

Batch Processing Statistics

Metric	Last batch
Processing Time	65 ms
Scheduling Delay	0 ms
Total Delay	65 ms

图 4. 51　Spark Streaming 应用的批处理统计信息

4. 4. 3　内存调优

内存调优方面的细节可以参考官方网站上的调优指南部分。这里主要讨论 Spark Streaming 应用相关的一些调优参数。

一个 Spark Streaming 应用对集群内存大小的需求，在很大程度上依赖于所使用的转换操作类型。比如，如果使用一个窗口长度至少为 10 分钟的窗口操作，那么集群内存必须足够装载这 10 分钟的数据。或者，如果对大量 Keys 进行 updateStateByKey 操作的话，对内存的要求也会极高。相反的，如果只是简单地做一种 map – filter – store 操作的话，对内存的需求是比较小的。

通常我们使用 StorageLevel. MEMORY_AND_DISK_SER_2 存储等级来对接收到的数据进行持久化，因此，当内存不足以装载数据时，会将数据溢出到磁盘中。这会降低流应用的性能，因此在流应用中，应提供足够的内存。

内存调优的另一个方面是垃圾回收 GC。一般流应用对延迟的要求很高，如果经常由于 GC 而导致大量停顿，这是不可接受的。

下面是一些可以对内存使用和 GC 开销进行调优的参数：

1. DStreams 的持久化级别：输入数据和 RDD 默认情况下是以序列化的字节进行持久化的。这可以同时减少内存使用和 GC 开销。可以参考前面的数据序列化部分。我们可以使用 Kryo 序列化来进一步减少序列化后的数据大小和内存使用。除了使用更好的序列化器，还可以加入压缩机制，可以参考配置属性 Spark. rdd. compress 的设置，以 CPU 的时间开销来交互内存使用和 GC 开销。

2. 旧数据的清除：默认情况下，输入的数据和 DStream 转换过程中产生的持久化的 RDD，都会自动被清除。Spark Streaming 会根据使用的转换操作来决定何时清除这些数据。比如窗口长度为 10 分钟的窗口操作，SparkStreaminig 会保留最后 10 分钟左右的数据，并积极丢弃旧的数据。可以通过配置参数 streamingContext. remember 为数据设置更长的保留时间。

3. CMS 垃圾收集器：这里强烈建议使用并行的 mark – and – sweep GC，以便持续地将 GC 相关的停顿保持中较低的状态。强烈建议在 Driver 和 Executor 侧都设置 CMS GC，可以分

别在 spark - submit 中使用 - - driver - java - options 设置 Driver，在属性配置中使用 spark. executor. extraJavaOptions 属性来设置 Executors。

4. 其他建议：为了进一步减少 GC 开销，可以尝试以下的方法：

1）持久化 RDD 时选择 off - heap 存储级别来使用 Tachyon。



2）使用更多个数的，分配的 heap sizes 更小的 executors。这可以在各个 JVM heap 中减少 GC 压力。

第5章 Tachyon 实践案例与解析

5.1 Tachyon 概述
5.2 重新编译部署包
5.3 Tachyon 部署的案例与解析
5.4 Tachyon 配置的案例与解析
5.5 命令行接口的案例与解析
5.6 同步底层文件系统的案例与解析
5.7 基于 Tachyon 运行的案例与解析



Section
5.1 Tachyon 概述

Tachyon 是 AMPLab 的李浩源所开发的一个基于内存的分布式文件系统，是 AMPLab 的 BDAS 的一个组成部分，在协议栈中的位置，如图 5.1 所示。

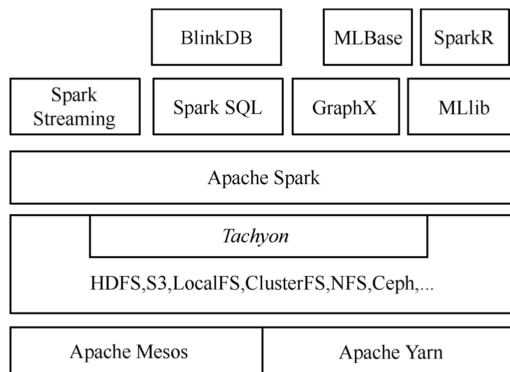


图 5.1　伯克利数据分析协议栈

Spark 内存计算框架，只是提供了强大的内存计算能力，并未提供存储能力。通过 Tachyon 内存分布式文件系统，可以为 Spark 等计算框架提供内存存储能力。不仅如此，Tachyon 内存分布式文件系统可以为集群中部署的计算框架提供内存级别的数据共享，即不同框架中的应用都可以基于 Tachyon 来共享数据。

Spark 计算框架内跨进程的数据共享：存储引擎和计算引擎在同一个进程（JVM）内，不同的 Jobs 之间共享数据是通过磁盘（disk）读写来实现的，如图 5.2 所示。

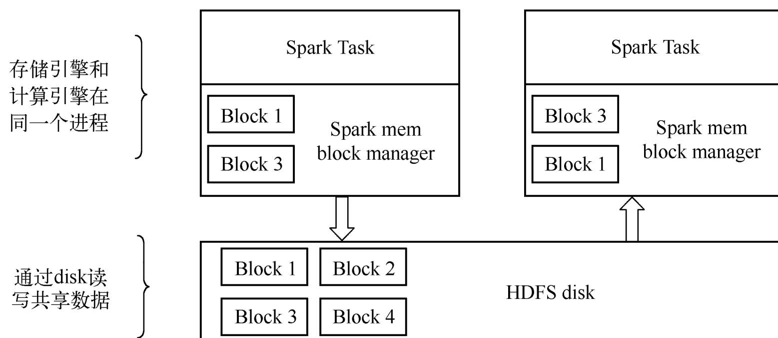


图 5.2　Spark 计算框架内跨进程的数据共享

Spark 计算框架与 Hadoop 计算框架间的数据共享也需要通过磁盘读写来实现，如图 5.3 所示。

基于 Tachyon，可以实现多个计算框架间的数据共享，如图 5.4 所示。

同时，单独抽离出一个内存分布式文件系统，还可以避免由于计算框架执行失败而导致数据丢失等问题，比如 Spark 内部进程崩溃（crash）而导致的内存数据丢失问题，如图 5.5 所示。

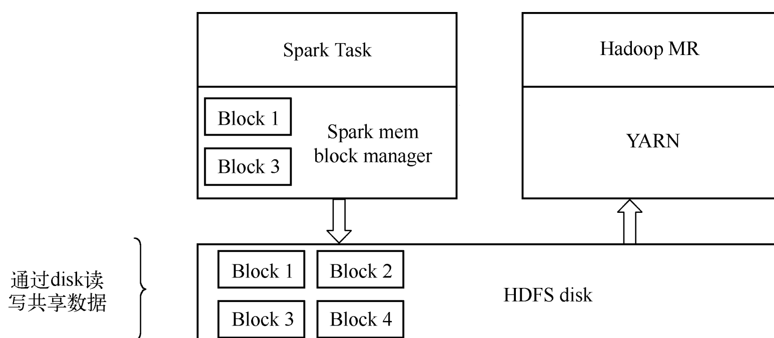


图 5.3　Spark 与 Hadoop 间的 disk 数据共享

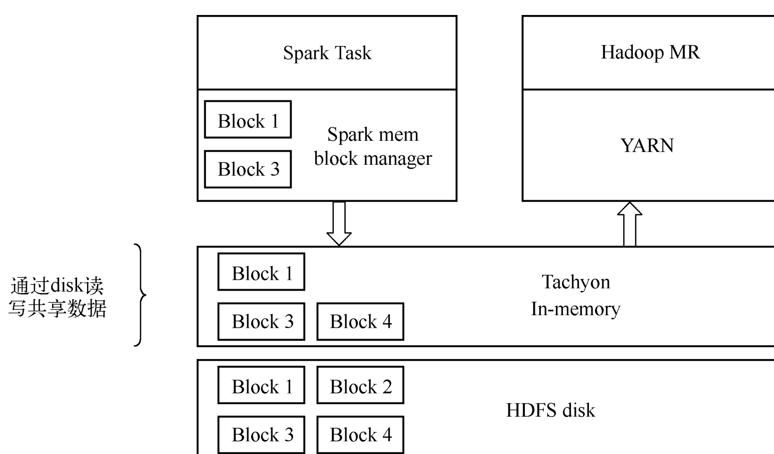


图 5.4　Spark 与 Hadoop 间的 Tachyon 数据共享

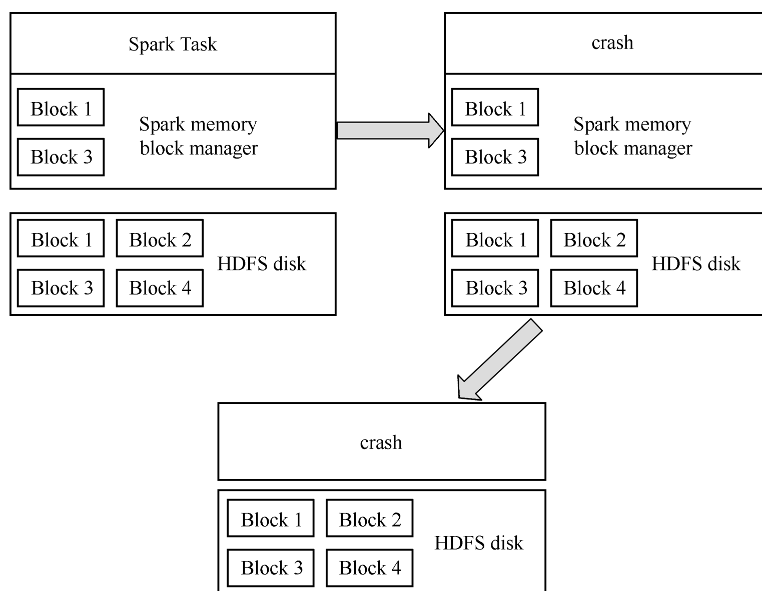


图 5.5　进程 crash 而导致的内存数据丢失

使用 Tachyon 不仅可以避免多个框架多个内存缓存，还避免了因进程崩溃（crash）而导致的数据丢失问题，同时还引入了堆外内存（off－heap memory）技术来降低 GC 的开销。



Section 5.2 重新编译部署包

在对案例进行分析之前，为了保证各个版本间的兼容性，需要重新编译 Tachyon 和 Spark 版本。

5.2.1 重新编译 Tachyon 的部署包

由于环境中使用了较新的 Hadoop 2.6.0 版本，因此需要对 Tachyon 进行重编译。首先下载 Tachyon 0.6.3 版本的源代码：

```
$ wget https://codeload.github.com/amplab/tachyon/tar.gz/v0.6.3
$ tar xvfz v0.6.3
$ cd tachyon-0.6.3
```

当前基于 Hadoop 2.6.0 的版本重新编译 Tachyon 0.6.3 版本，构建部署包：

```
[root@ cluster01 tachyon-0.6.3]#mvn package  -DskipTests  -Dhadoop.version=2.6.0
```

当出现以下输出信息时，部署包构建成功。

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Tachyon ProjectParent ............................ SUCCESS [6.465s]
[INFO] Tachyon ProjectCore .............................. SUCCESS [49.772s]
[INFO] Tachyon ProjectClient ............................ SUCCESS [8.358s]
[INFO] Tachyon ProjectAssemblies ........................ SUCCESS [0.975s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:1:05.867s
[INFO] Finished at:Fri Apr 17 10:37:04 PDT 2015

[INFO] Final Memory:37M/448M
```

5.2.2 重新编译 Spark 的部署包

Spark 官网提供的 Tachyon 版本不是最新的 Tachyon 0.6.3 版本，因此，根据当前环境，基于 Tachyon 0.6.3 版本和 Hadoop 2.6.0 版本重新编译部署包。

1. Tachyon 版本的修改需要修改 pom. xml 和代码。

1）修改 core/pom. xml 中 Tachyon 的版本为 0. 6. 3。

```
<groupId>org.tachyonproject</groupId>
<artifactId>tachyon-client</artifactId>
<version>0.6.3</version>
……
```

2）同时修改代码，修改类的路径如下：

```
spark1.3.0/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala
```

3）修改前的编译错误如下：

```
[ERROR] /harli/spark-1.3.0/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala:67:type mismatch;
found   :String
required:tachyon.TachyonURI
[ERROR]       client.exist(file.getPath())
[ERROR]                                         ^
[ERROR] /harli/spark-1.3.0/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala:85:type mismatch;
found   :String
required:tachyon.TachyonURI
[ERROR]              client.mkdir(path)
[ERROR]                                 ^
[ERROR] /harli/spark-1.3.0/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala:93:type mismatch;
found   :String
required:tachyon.TachyonURI
[ERROR]       if(!client.exist(filePath)){
[ERROR]                                ^
[ERROR] /harli/spark-1.3.0/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala:117:type mismatch;
found   :String
required:tachyon.TachyonURI
[ERROR]              if (!client.exist(path)){
[ERROR]                                       ^
[ERROR] /harli/spark-1.3.0/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala:118:type mismatch;
found   :String
required:tachyon.TachyonURI
[ERROR]               foundLocalDir = client.mkdir(path)
[ERROR]
```

由于 Tachyon 0. 6. 3 版本对外提供的文件系统访问接口的变更，导致 Spark 1. 3 的访问代码编译失败。根据接口的变更，将其中所有 String 类型的 path 等信息修改为 new TachyonURI (path)等，同时在文件头部添加导入语句：import tachyon. TachyonURI。

使用 diff 工具查看修改前后的文件差异，修改后的差异如下（Spark-1. 3. 0. new 是修改

后的)：

```
[root@ cluster01 harli]# diff ./spark-1.3.0/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala ./spark-1.3.0.new/core/src/main/scala/org/apache/spark/storage/TachyonBlockManager.scala
24a25
> import tachyon.TachyonURI;
67c68
<       client.exist(file.getPath())
---
>       client.exist(newTachyonURI(file.getPath()))
85c86
<           client.mkdir(path)
---
>           client.mkdir(newTachyonURI(path))
93c94
<       if(!client.exist(filePath)) {
---
>       if(!client.exist(newTachyonURI(filePath))) {
117,118c118,119
<           if (!client.exist(path)) {
< foundLocalDir = client.mkdir(path)
---
>           if (!client.exist(newTachyonURI(path))) {
> foundLocalDir = client.mkdir(new TachyonURI(path))
```

2. 修改后，使用部署脚本 make-distribution.sh 重新编译、部署，命令如下：

```
[root@ cluster01 spark-1.3.0]# ./make-distribution.sh --skip-java-test --tgz --mvnmvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.6.0 -Phive -Dhive.version=0.13.1 -Phive -thriftserver
```

3. 出现以下信息时，表示编译成功。

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Spark Project ParentPOM .......................... SUCCESS [4.532s]
[INFO] Spark ProjectNetworking .......................... SUCCESS [9.115s]
[INFO] Spark Project Shuffle StreamingService ........... SUCCESS [4.977s]
[INFO] Spark ProjectCore ................................ SUCCESS [3:18.231s]
[INFO] Spark ProjectBagel ............................... SUCCESS [18.997s]
[INFO] Spark ProjectGraphX .............................. SUCCESS [52.297s]
[INFO] Spark ProjectStreaming ........................... SUCCESS [1:15.737s]
[INFO] Spark ProjectCatalyst ............................ SUCCESS [1:21.429s]
[INFO] Spark ProjectSQL ................................. SUCCESS [1:39.423s]
[INFO] Spark Project MLLibrary .......................... SUCCESS [1:50.423s]
[INFO] Spark ProjectTools ............................... SUCCESS [11.017s]
[INFO] Spark ProjectHive ................................ SUCCESS [1:16.486s]
[INFO] Spark ProjectREPL ................................ SUCCESS [33.244s]
```

```
[INFO] Spark ProjectYARN ................................ SUCCESS [39.287s]
[INFO] Spark Project Hive ThriftServer .................. SUCCESS [24.989s]
[INFO] Spark ProjectAssembly ............................ SUCCESS [1:49.777s]
[INFO] Spark Project ExternalTwitter .................... SUCCESS [15.131s]
[INFO] Spark Project External FlumeSink ................. SUCCESS [17.302s]
[INFO] Spark Project ExternalFlume ...................... SUCCESS [23.973s]
[INFO] Spark Project ExternalMQTT ....................... SUCCESS [15.707s]
[INFO] Spark Project ExternalZeroMQ ..................... SUCCESS [16.107s]
[INFO] Spark Project ExternalKafka ...................... SUCCESS [33.137s]
[INFO] Spark ProjectExamples ............................ SUCCESS [2:06.255s]
[INFO] Spark Project YARN ShuffleService ................ SUCCESS [19.570s]
[INFO] Spark Project External KafkaAssembly ............. SUCCESS [34.164s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:20:52.068s
[INFO] Finished at:Sun Apr 19 07:53:17 PDT 2015
[INFO] Final Memory:93M/1170M
[INFO] ------------------------------------------------------------------------
```

由于 Maven 仓库（http://maven.oschina.net）中缺少一些依赖包，需要自己手动下载，或修改相关的版本依赖信息，或用其他仓库。比如 org.eclipse.paho.client.mqttv3 的 1.0.1 版本和 Hive 1.1.0 版本的依赖包在仓库中不存在，如图 5.6 所示。



图 5.6　mqttv3 的 1.0.1 版本缺失

仓库中的 Hive 依赖版本信息，如图 5.7 所示。



图 5.7　Hive 的 1.1.0 版本缺失

这里手动下载了 org. eclipse. paho. client. mqttv3 的 1. 0. 1 版本，直接使用 Hive 的 0. 13. 1 版本。手动下载后放置的目录根据 jar 包的坐标放置，如 org. eclipse. paho. client. mqttv3 的 jar 包在本机中放置的路径为：. m2/repository/org/eclipse/paho/org. eclipse. paho. client. mqttv3/1. 0. 1。

Section 5. 3 Tachyon 部署的案例与解析

一般情况下，分布式系统都会至少提供两种部署模式，一种是单机模式，通常用于测试、快速部署入门等；另一种是分布式模式，用于实际工作环境。比如 Hadoop 分布式系统，单机模式和伪分布式模式都是为了用于测试和快速部署入门，其中伪分布式以进程来模拟集群节点。类似的，Spark 也是可以在单机上手动启动 Worker 守护进程，来模拟伪分布式的 Spark 集群。

Tachyon 也同时提供了这两种部署模式，这里会分别给出详细的部署案例并进行解析。集群中部署的各软件版本如下：

1）Tachyon 0. 6. 3。

2）Hadoop 2. 6. 0。

3）Spark 1. 3. 0。

为了保证各软件版本间的兼容性，针对这些版本进行了重编译，其中 Tachyon 和 Spark 的重新编译参看 5. 2. 2 小节的内容。

5. 3. 1 单机模式部署的案例与解析

一、单机环境

1. 当前用户名 harli；如果是 root 用户，中间可以省去 sudo 权限验证。
2. 当前单机的 hostname 为 cluster04。

二、准备工作

启动单机模式下的 Tachyon 之前，需要先对环境变量进行配置，步骤如下：

```
[harli@cluster04 tachyon]$ mv tachyon-0.6.3/ tachyon
[harli@cluster04 cluster]$ cd tachyon
[harli@cluster04 tachyon]$ cp conf/tachyon-env.sh.template conf/tachyon-env.sh
[harli@cluster04 tachyon]$ ./bin/tachyon format
Connection to localhost asharli...  Warning: Permanently added localhost (RSA) to the list of known hosts.
which: no java in (/usr/local/bin:/bin:/usr/bin)
dirname: missing operand
Try dirname --help for more information.
Formatting Tachyon Worker @ cluster04
/home/harli/cluster/tachyon/bin/tachyon: line 243: /../bin/java: No such file or directory
Connection to localhost closed.
Formatting Tachyon Master @ localhost
[harli@cluster04 tachyon]$
```

找不到 Java 命令，因此添加 JAVA_HOME 设置。

```
[harli@cluster04 tachyon]$ echo $JAVA_HOME
/lib/jdk1.7.0_71
[harli@cluster04 tachyon]$ vim conf/tachyon-env.sh
```

添加后的内容如下：

```
JAVA_HOME="/lib/jdk1.7.0_71"
if [ -z "$JAVA_HOME" ]; then
  export JAVA_HOME="$(dirname $(which java))/.."
fi
```

其中斜体部分是添加的环境变量。

```
[harli@cluster04 tachyon]$ ./bin/tachyon format
Connection to localhost asharli... Formatting Tachyon Worker @ cluster04
Connection to localhost closed.
Formatting Tachyon Master @ localhost
```

local 模式下启动 Tachyon：

```
[harli@cluster04 tachyon]$ ./bin/tachyon-start.sh local
Killed 0 processes
Killed 0 processes
Connection to localhost asharli... Killed 0 processes
Connection to localhost closed.
We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:
    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great powercomes great responsibility.
[sudo] password forharli:
harli is not in the sudoers file.  This incident will be reported.
Mount failed,not starting
```

首先需要对 Tachyon 的文件系统进行格式，需要注意的是，由于 Tachyon 需要安装 ramfs，启动 local 模式时，用户需要有权限进行操作。通常可以使用 root 用户，或能为其他用户添加 sudo 权限。

```
[root@cluster04 ~]# su - root
[root@cluster04 ~]#chmod u+w /etc/sudoers
[root@cluster04 ~]# vim /etc/sudoers
[root@cluster04 ~]#chmod u-w /etc/sudoers
```

修改文件/etc/sudoers 时，添加 harli 用户（当前启动 Tachyon 的用户）：

```
## Allow root to run any commands anywhere
root     ALL=(ALL)       ALL
harli    ALL=(ALL)       ALL
#harli   ALL=(root) NOPASSWD:ALL
```

在 root 用户下，添加相同权限的 harli 用户

小技巧：如果不想在 sudo 过程中进行权限验证的话，可以给用户设置无密码 sudo 权限。另外 chmod 修改权限部分也可以通过 vim 修改文件内容时，使 w！命令强制写入来简化。无密设置后可以通过下面命令来测试：



```
//无密设置后不需要再进行权限验证
[harli@ cluster04 ~]$ sudo ls /root/
anaconda - ks. cfg   install. log   install. log. syslog   test

[harli@ cluster04 ~]$ ls /root/
ls:cannot open directory /root/:Permission denied
```

三、启动 Tachyon

local 模式启动 Tachyon：

```
[harli@ cluster04 tachyon]$ ./bin/tachyon - start. sh local
Killed 0 processes
Killed 0 processes
Connection to localhost asharli...  Killed 0 processes
Connection to localhost closed.
[sudo] password forharli:
Sorry,try again.
[sudo] password forharli:
ERROR:Memory(1028517888) is less than requested ramdisk size(1073741824).
Please reduce Tachyon_WORKER_MEMORY_SIZE
Mount failed,not starting
```

由于当前内存值设置太小，导致 ramdisk 大小不足报错。

查看环境配置文件中的相关属性：

```
[harli@ cluster04 tachyon]$ vim conf/tachyon - env. sh
```

找到 Tachyon_WORKER_MEMORY_SIZE 配置，默认值为 1 GB（block 大小）：

```
export Tachyon_WORKER_MEMORY_SIZE = 1 GB
```

修改当前虚拟机的内存，内存空间由原来的 1 GB 改为 3 GB，具体修改如图 5. 8 所示。

重新启动单机模式下的 Tachyon：

```
[harli@ cluster04 tachyon]$ ./bin/tachyon - start. sh local
Killed 0 processes
Killed 0 processes
Connection to localhost asharli...  Killed 0 processes
Connection to localhost closed.
[sudo] password forharli:
FormattingRamFS:/mnt/ramdisk (1gb)
Starting master @ localhost
Starting worker @ cluster04
```



图 5.8　虚拟机内存设置的修改

启动成功，查看当前进程：

```
[harli@ cluster04 tachyon]$ jps
4386TachyonMaster
4405TachyonWorker
4451 Jps
```

查看 mount 信息：

```
[harli@ cluster04 tachyon]$ mount | grep ramdisk
ramfs on /mnt/ramdisk type ramfs (rw,size=1gb)
```

可以看到当前新增了一个映射 ramdisk，大小为默认值 1GB。

四、验证

查看 Web Interface 界面（http：cluster04：19999），如图 5.9 所示。

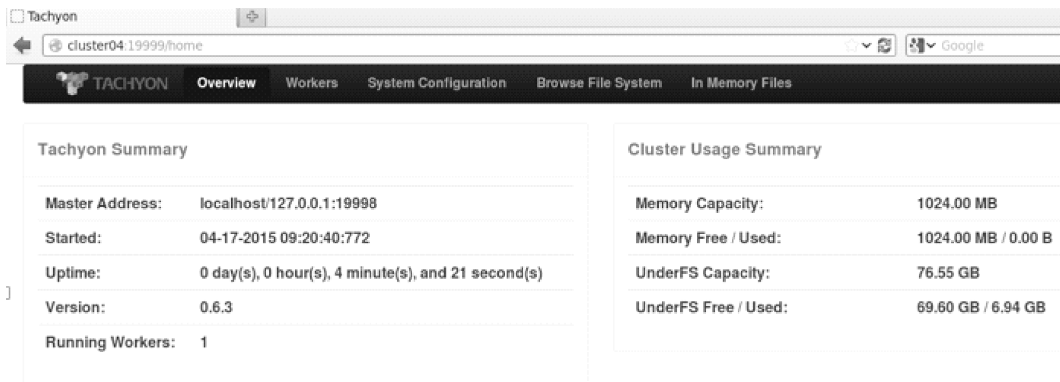


图 5.9　Tachyon 的概览页面

Web Interface 界面包含以下几部分内容：

1）Overview：Tachyon 集群的整体描述信息，包含当前运行的 TachyonURL 信息，运行



中的 Workers 个数，以及当前集群使用情况。

2）Workers：当前可用 Workers 及其相关信息，以及当前丢失的 Workers 信息。

3）System Configuration：当前 Tachyon 集群的系统配置信息。

4）BrowseFileSystem：Tachyon 文件系统的浏览界面。

5）In Memory Files：当前在内存中的文件信息。

到这一步单机模式已经部署成功。如果启动失败，可以查看启动日志，日志目录在 logs 下，当前日志包含：

```
[harli@cluster04 tachyon]$ ls logs/
master.log@192.168.242.135_04-17-2015    user.log@192.168.242.135_04-17-2015
user.log@192.168.242.135_04-17-2015_2    worker.out
master.out                               user.log@192.168.242.135_04-17-2015_1
worker.log@192.168.242.135_04-17-2015
```

运行一个简单的测试：

```
[harli@cluster04 tachyon]$ ./bin/tachyonrunTest Basic CACHE_THROUGH
/default_tests_files/BasicFile_CACHE_THROUGH has been removed
2015-04-17 09:30:59,415 WARN  (CommonConf.java:<init>) - tachyon.home is not
set. Using /mnt/tachyon_default_home as the default value.
2015-04-17 09:30:59,430 INFO  (MasterClient.java:connect) - Tachyon client
(version 0.6.3) is trying to connect master @ localhost/127.0.0.1:19998
2015-04-17 09:30:59,468 INFO  (MasterClient.java:connect) - User registered
at the master localhost/127.0.0.1:19998 got UserId 3
2015-04-17 09:30:59,596 INFO  (CommonUtils.java:printTimeTakenMs) -
createFile withfileId 3 took 177 ms.
2015-04-17 09:30:59,644 INFO  (WorkerClient.java:connect) - Trying to get
local worker host:cluster04
2015-04-17 09:30:59,672 INFO  (WorkerClient.java:connect) - Connecting local
worker @ cluster04/192.168.242.135:29998
2015-04-17 09:30:59,792 INFO  (TachyonFS.java:getLocalBlockTemporaryPath) -
Folder /mnt/ramdisk/tachyonworker/users/3 was created!
2015-04-17 09:30:59,795 INFO  (BlockOutStream.java:<init>) -
/mnt/ramdisk/tachyonworker/users/3/3221225472 was created!
2015-04-17 09:30:59,872 INFO  (CommonUtils.java:printTimeTakenMs) - writeFile
to file /default_tests_files/BasicFile_CACHE_THROUGH took 276 ms.
2015-04-17 09:31:00,012 INFO  (CommonUtils.java:printTimeTakenMs) - readFile
file /default_tests_files/BasicFile_CACHE_THROUGH took 139 ms.
Passed the test!
```

出现以上信息表示验证通过。

继续查看 Web Interface 界面。

1. Overview 页面：当前使用的内存已经发生变化

当前使用内存的变化如图 5.10 所示。

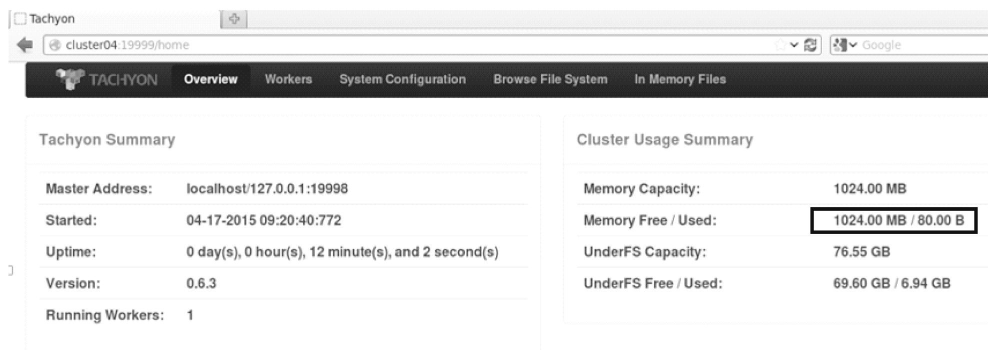


图 5.10　Tachyon 的概览页面上的内存变化

2. Workers

Workers 页面上的相应变化如图 5.11 所示。

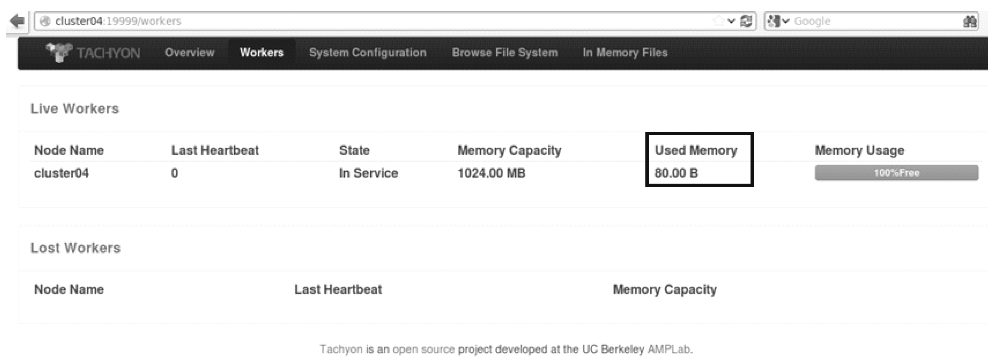


图 5.11　Tachyon 的 Workers 页面上的变化

3. Browse File System

Tachyon 的文件系统页面上的相应变化如图 5.12 所示。



图 5.12　Tachyon 的文件系统页面上的变化

4. In Memory Files

缓存在内存中的文件的界面如图 5.13 所示。

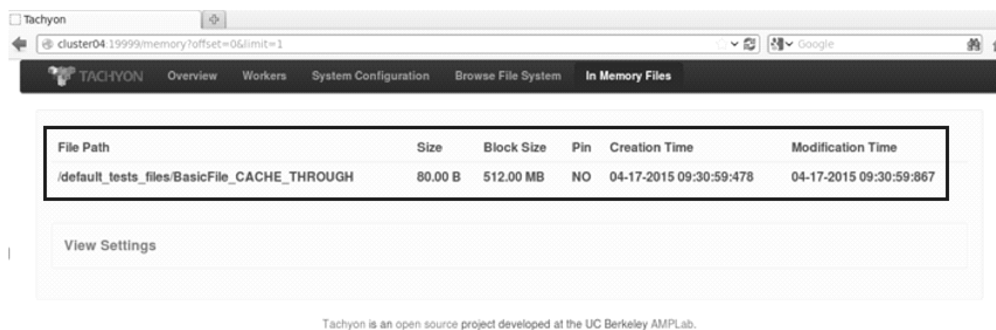


图 5.13　Tachyon 的缓存在内存中文件的界面

继续单击文件名，对应的文件具体内容如图 5.14 所示。



图 5.14　查看 Tachyon 文件

图 5.14 中显示乱码，应该和本机编码默认为 UTF－8 有关，当前编码为：

```
[harli@ cluster04 tachyon] $  locale
LANG = en_US. UTF－8
LC_CTYPE = "en_US. UTF－8"
LC_NUMERIC = "en_US. UTF－8"
LC_TIME = "en_US. UTF－8"
LC_COLLATE = "en_US. UTF－8"
LC_MONETARY = "en_US. UTF－8"
LC_MESSAGES = "en_US. UTF－8"
LC_PAPER = "en_US. UTF－8"
LC_NAME = "en_US. UTF－8"
```

```
LC_ADDRESS = "en_US. UTF - 8"
LC_TELEPHONE = "en_US. UTF - 8"
LC_MEASUREMENT = "en_US. UTF - 8"
LC_IDENTIFICATION = "en_US. UTF - 8"
LC_ALL =
```

可以继续运行全部的测试：

```
[harli@ cluster04 tachyon] $ ./bin/tachyonrunTests
/home/harli/cluster/tachyon/bin/tachyon runTest Basic MUST_CACHE
/default_tests_files/BasicFile_MUST_CACHE has been removed
2015 - 04 - 17 09:43:35,285 WARN   (CommonConf. java:<init>) - tachyon. home is not
set. Using /mnt/tachyon_default_home as the default value.
2015 - 04 - 17 09:43:35,300 INFO   (MasterClient. java:connect) - Tachyon client
(version 0.6.3) is trying to connect master @ localhost/127.0.0.1:19998
2015 - 04 - 17 09:4:35,337 INFO   (MasterClient. java:connect) - User registered
at the master localhost/127.0.0.1:19998 got UserId 7
2015 - 04 - 17 09:43:35,352 INFO   (CommonUtils. java:printTimeTakenMs) -
createFile withfileId 4 took 62 ms.
2015 - 04 - 17 09:43:35,386 INFO   (WorkerClient. java:connect) - Trying to get
local worker host:cluster04
2015 - 04 - 17 09:43:35,394 INFO   (WorkerClient. java:connect) - Connecting local
worker @ cluster04/192.168.242.135:29998
2015 - 04 - 17 09:43:35,431 INFO   (TachyonFS. java:getLocalBlockTemporaryPath) -
Folder /mnt/ramdisk/tachyonworker/users/7 was created!
2015 - 04 - 17 09:43:35,439 INFO   (BlockOutStream. java:<init>) -
/mnt/ramdisk/tachyonworker/users/7/4294967296 was created!
2015 - 04 - 17 09:43:35,470 INFO   (CommonUtils. java:printTimeTakenMs) - writeFile
to file /default_tests_files/BasicFile_MUST_CACHE took 118 ms.
2015 - 04 - 17 09:43:35,531 INFO   (CommonUtils. java:printTimeTakenMs) - readFile
file /default_tests_files/BasicFile_MUST_CACHE took 61 ms.
Passed the test!
……..
```

之后查看 Web Interface 页面，这里查看 In Memory 页面，如图 5.15 所示。

这是测试后映射的 ramdisk 中的内容：

```
[harli@ cluster04 tachyon] $ cd /mnt/ramdisk/tachyonworker/
100931731456   105226698752   108447924224   24696061952    33285996544
52613349376    68719476736    90194313216    97710505984
102005473280   106300440576   109521666048   25769803776    4294967296
53687091200    74088185856    95563022336    99857989632
103079215104   10737418240    110595407872   31138512896    46170898432
54760833024    75161927680    9663676416                    users/
104152956928   107374182400   11811160064    32212254720    47244640256
67645734912    76235669504    96636764160
```

单机模式下可以作为入门测试等场景使用，最终在实际环境上应该部署集群模式。这里先停止单机模式，命令如下：

File Path	Size	Block Size	Pin	Creation Time	Modification Time
/default_tests_files/BasicCheckpoint/part-0	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:284	04-17-2015 09:43:51:375
/default_tests_files/BasicCheckpoint/part-1	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:285	04-17-2015 09:43:51:382
/default_tests_files/BasicCheckpoint/part-2	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:285	04-17-2015 09:43:51:400
/default_tests_files/BasicCheckpoint/part-3	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:286	04-17-2015 09:43:51:405
/default_tests_files/BasicCheckpoint/part-4	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:286	04-17-2015 09:43:51:411
/default_tests_files/BasicCheckpoint/part-5	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:286	04-17-2015 09:43:51:417
/default_tests_files/BasicCheckpoint/part-6	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:286	04-17-2015 09:43:51:423
/default_tests_files/BasicCheckpoint/part-7	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:286	04-17-2015 09:43:51:433
/default_tests_files/BasicCheckpoint/part-8	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:286	04-17-2015 09:43:51:439
/default_tests_files/BasicCheckpoint/part-9	80.00 B	512.00 MB	NO	04-17-2015 09:43:51:287	04-17-2015 09:43:51:447
/default_tests_files/BasicFile_ASYNC_THROUGH	80.00 B	512.00 MB	NO	04-17-2015 09:43:49:535	04-17-2015 09:43:49:641
/default_tests_files/BasicFile_CACHE_THROUGH	80.00 B	512.00 MB	NO	04-17-2015 09:43:42:274	04-17-2015 09:43:42:396
/default_tests_files/BasicFile_MUST_CACHE	80.00 B	512.00 MB	NO	04-17-2015 09:43:35:340	04-17-2015 09:43:35:463

图 5.15　Tachyon 全部测试后的内存文件信息

```
[harli@cluster04 tachyon]$ ./bin/tachyon-stop.sh
Killed 1 processes
Killed 1 processes
Connection to localhost asharli... Killed 0 processes
Connection to localhost closed.
```

至此，完成单机模式部署的实践。

5.3.2 集群模式部署的案例与解析

这里分析 Standalone 集群的搭建过程。

一、集群环境

1）当前用户名 harli。如果是 root 用户，中间可以省去 sudo 权限验证。

2）集群机器。包括 cluster04、cluster05、cluster06 三个节点。

3）当前集群的进程如下。

节　点	进　程
cluster04	TachyonMaster、TachyonWorker、Master、NameNode、DataNode、Worker
cluster05	TachyonWorker、QuorumPeerMain、DataNode、Worker
cluster06	TachyonMaster、TachyonWorker、DataNode、Worker

二、准备工作

1. 修改 conf/workers，添加以下三个节点

```
cluster04
cluster05
cluster06
```

需要注意的是 Tachyon 0. 5. 0 等旧版本中配置文件名字为 slaves。

2. 修改 conf/tachyon - env. sh

和单机部署一样，需要添加 JAVA_HOME 环境变量。

```
JAVA_HOME = "/lib/jdk1. 7. 0_71"
if [  -z " $ JAVA_HOME" ]; then
  export JAVA_HOME = " $ (dirname  $ (which java))/.. "
fi
export JAVA = " $ JAVA_HOME/bin/java"
export Tachyon_MASTER_ADDRESS = cluster04
```

除了 JAVA_HOME 环境变量之前，尤其要注意的是要修改 Tachyon_MASTER_ADDRESS 为本机地址，否则 Workers 连接 Master 时会使用默认的 localhost，导致连接失败。

3. 修改完成后，发布到各个 Workers 节点上，这里使用 scp 命令

```
[harli@ cluster04 tachyon] $ scp  -r ../tachyon harli@ cluster05:/home/harli/cluster
[harli@ cluster04 tachyon] $ scp  -r ../tachyon harli@ cluster05:/home/harli/cluster
```

4. 格式化

```
[harli@ cluster04 tachyon] $ ./bin/tachyon format
Connection to cluster04 asharli... Formatting Tachyon Worker @ cluster04
Connection to cluster04 closed.
Connection to cluster05 asharli... Formatting Tachyon Worker @ cluster05
Connection to cluster05 closed.
Connection to cluster06 asharli... Formatting Tachyon Worker @ cluster06
Connection to cluster06 closed.
Formatting Tachyon Master @ cluster04
```

5. 启动

```
[harli@ cluster04 tachyon] $ ./bin/tachyon - start. sh all Mount
Killed 0 processes
Killed 0 processes
Connection to cluster04 asharli... Killed 0 processes
Connection to cluster04 closed.
Connection to cluster05 asharli... Killed 0 processes
Connection to cluster05 closed.
Connection to cluster06 asharli... Killed 0 processes
Connection to cluster06 closed.
Starting master @ localhost
Connection to cluster04 asharli... Formatting RamFS:/mnt/ramdisk (1gb)
umount:only root can do that
mount:only root can do that
chmod:changing permissions of '/mnt/ramdisk':Operation not permitted
Mount failed,not starting worker
Connection to cluster04 closed.
Connection to cluster05 asharli... Formatting RamFS:/mnt/ramdisk (1gb)
mkdir:cannot create directory '/mnt/ramdisk':Permission denied
```

```
mount:only root can do that
chmod:cannot access' /mnt/ramdisk' :No such file or directory
Mount failed,not starting worker
Connection to cluster05 closed.
Connection to cluster06 asharli... Formatting RamFS:/mnt/ramdisk (1gb)
mkdir:cannot create directory' /mnt/ramdisk' :Permission denied
mount:only root can do that
chmod:cannot access' /mnt/ramdisk' :No such file or directory
Mount failed,not starting worker
Connection to cluster06 closed.
```

当使用 all Mount 方式进行启动时，必须是 root 用户才可以直接进行 ramdisk 的映射。

由于这里使用 harli 用户，因此，除了在各个 Workers 上设置 harli 用户的 sudo 权限之外，在启动时，参数必须选择 SudoMount，对应的，如果是 root 用户，参数设为 mount 即可。

使用正确的参数启动：

```
[harli@ cluster04 tachyon] $ ./bin/tachyon-start.sh allSudoMount
Killed 1 processes
Killed 1 processes
Connection to cluster04 asharli... Killed 0 processes
Connection to cluster04 closed.
Connection to cluster05 asharli... Killed 1 processes
Connection to cluster05 closed.
Connection to cluster06 asharli... Killed 1 processes
Connection to cluster06 closed.
Starting master @ cluster04
Connection to cluster04 asharli... [sudo] password for harli:
FormattingRamFS:/mnt/ramdisk (1gb)
Starting worker @ cluster04
Connection to cluster04 closed.
Connection to cluster05 asharli... [sudo] password for harli:
FormattingRamFS:/mnt/ramdisk (1gb)
Starting worker @ cluster05
Connection to cluster05 closed.
Connection to cluster06 asharli... [sudo] password for harli:
FormattingRamFS:/mnt/ramdisk (1gb)
Starting worker @ cluster06
Connection to cluster06 closed.
```

注意：当使用非 root 用户时，在每个主机上映射 ramdisk 时都要进行 sudo 权限验证，会非常麻烦。

6. 查看进程

使用 jps 命令查看各个节点当前进程情况：

节　点	进　程
cluster04	TachyonMaster、TachyonWorker
cluster05	TachyonWorker
cluster06	TachyonWorker

查看 Web Interface 界面（http://cluster04:19999），如图 5.16 所示。

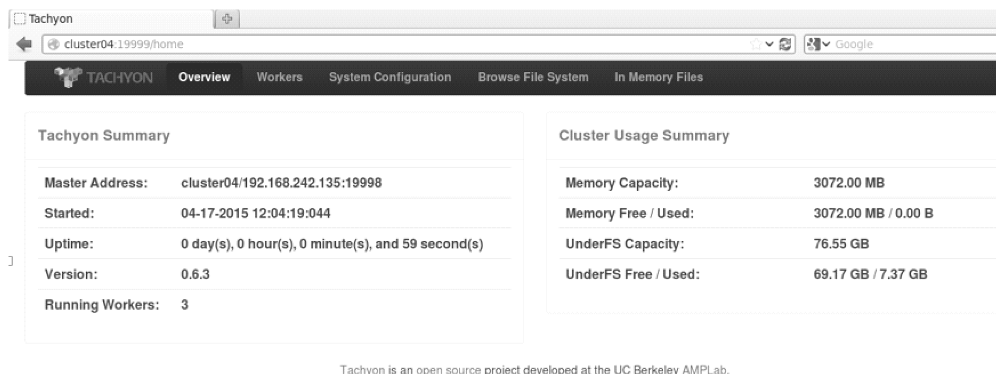


图 5.16　Tachyon 启动后的验证界面

可以看到已经可以成功访问页面，并且运行中的 Workers 节点个数为 3。

继续运行测试：

```
[harli@cluster04 tachyon]$ ./bin/tachyon runTests
```

再次查看界面，如图 5.17 所示。

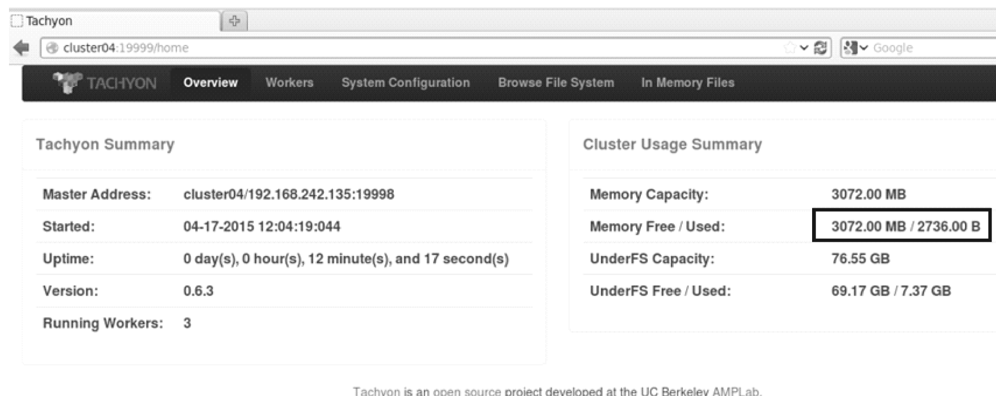


图 5.17　Tachyon 运行测试后的界面

验证通过，至此，集群模式部署成功。

5.3.3　集群 Master 容错部署的案例与解析

作为一个 M/S 架构的分布式系统，普遍存在单点故障问题，在 Tachyon 分布式系统中，对于 Master 节点的单点故障也是采用 ZooKeeper 解决，即 Tachyon 的容错是基于多 Master 方法，通过 ZooKeeper 集群管理进来管理 Master，负责选出一个 Master 作为 Leader，作为 Workers 和 Clients 访问的主入口，其他 Masters 作为备用（Standby），使用共享的 Journal 来确保和 Leader 具有相同的文件系统元数据，并在 Leader 宕机时快速接手。当作为 Leader 的 Master 宕机时会自动选举出新的 Leader，并且 Workers 和 Clients 也会自动连接到新的

Leader 上。

实现 Master 容错的前提条件：

1）搭建一个 ZooKeeper 环境，用户维护管理多个 Master。

2）选择一个共享的、可靠的底层文件系统来存放 Journal。



当前可以使用的底层文件系统有 HDFS、Amazon S3 和 GlusterFS，这里选择 HDFS 作为底层文件系统进行案例实践。

一、环境准备

针对 ZooKeeper 环境，这里使用 Flume 提供的 ZooKeeper，ZooKeeper 的启动可以参考第 4.3.3 小节的内容处理 Kafka 数据源的实践准备部分。同时要确保在环境中已经安装好了 Hadoop 环境。

二、环境配置

1. 修改配置文件 conf/tachyon-env.sh，添加或修改下面三个环境变量

```
JAVA_HOME="/lib/jdk1.7.0_71"
export Tachyon_MASTER_ADDRESS=cluster04
export Tachyon_UNDERFS_ADDRESS=hdfs://cluster04:9000
```

其中：

1）JAVA_HOME 为本地 java 安装目录。

2）Tachyon_MASTER_ADDRESS 为 Master 对外可见的地址，这里设置为 Master 的 hostname，即 cluster04。

3）Tachyon_UNDERFS_ADDRESS 为底层文件系统，这里使用 HDFS，HDFS 底层文件系统的配置可以参考第 5.4.1 小节的内容。

2. 修改配置文件 conf/tachyon-env.sh，添加或修改属性配置

修改前的属性配置：

```
export Tachyon_JAVA_OPTS+="
  -Dlog4j.configuration=file:$CONF_DIR/log4j.properties
  -Dtachyon.debug=false
  -Dtachyon.worker.hierarchystore.level.max=1
  -Dtachyon.worker.hierarchystore.level0.alias=MEM
  -Dtachyon.worker.hierarchystore.level0.dirs.path=$Tachyon_RAM_FOLDER
 -Dtachyon.worker.hierarchystore.level0.dirs.quota=$Tachyon_WORKER_MEMORY_SIZE
  -Dtachyon.underfs.address=$Tachyon_UNDERFS_ADDRESS
  -Dtachyon.underfs.hdfs.impl=$Tachyon_UNDERFS_HDFS_IMPL
  -Dtachyon.data.folder=$Tachyon_UNDERFS_ADDRESS/tmp/tachyon/data
  -Dtachyon.workers.folder=$Tachyon_UNDERFS_ADDRESS/tmp/tachyon/workers
  -Dtachyon.worker.memory.size=$Tachyon_WORKER_MEMORY_SIZE
  -Dtachyon.worker.data.folder=/tachyonworker/
  -Dtachyon.master.worker.timeout.ms=60000
  -Dtachyon.master.hostname=$Tachyon_MASTER_ADDRESS
  -Dtachyon.master.journal.folder=$Tachyon_HOME/journal/
  -Dorg.apache.jasper.compiler.disablejsr199=true
  -Djava.net.preferIPv4Stack=true
```

修改后的属性配置：

```
export Tachyon_JAVA_OPTS + = "
  - Dlog4j. configuration = file: $ CONF_DIR/log4j. properties
  - Dtachyon. debug = false
  - Dtachyon. worker. hierarchystore. level. max = 1
  - Dtachyon. worker. hierarchystore. level0. alias = MEM
  - Dtachyon. worker. hierarchystore. level0. dirs. path = $ Tachyon_RAM_FOLDER
 - Dtachyon. worker. hierarchystore. level0. dirs. quota = $ Tachyon_WORKER_MEMORY_SIZE
  - Dtachyon. underfs. address = $ Tachyon_UNDERFS_ADDRESS
  - Dtachyon. underfs. hdfs. impl = $ Tachyon_UNDERFS_HDFS_IMPL
  - Dtachyon. data. folder = $ Tachyon_UNDERFS_ADDRESS/tmp/tachyon/data
  - Dtachyon. workers. folder = $ Tachyon_UNDERFS_ADDRESS/tmp/tachyon/workers
  - Dtachyon. worker. memory. size = $ Tachyon_WORKER_MEMORY_SIZE
  - Dtachyon. worker. data. folder = /tachyonworker/
  - Dtachyon. master. worker. timeout. ms = 60000
  - Dtachyon. master. hostname = $ Tachyon_MASTER_ADDRESS
  - Dtachyon. master. journal. folder = $ Tachyon_UNDERFS_ADDRESS/journal/
  - Dorg. apache. jasper. compiler. disablejsr199 = true
  - Djava. net. preferIPv4Stack = true
  - Dtachyon. usezookeeper = true
  - Dtachyon. zookeeper. address = cluster05:2181
```

其中：

1）修改 tachyon. master. journal. folder 配置属性，将 journal 的目录放置于 HDFS 上。

2）增加 tachyon. usezookeeper 和 tachyon. zookeeper. address 属性配置，根据当前实际的 ZooKeeper 进行设置。

这里的 tachyon. data. folder 和 tachyon. workers. folder 可以保持不变。

修改完成后，需要将该配置文件拷贝到集群中的其他节点，命令如下：

```
[harli@ cluster04 tachyon] $ scp ./conf/tachyon - env. sh harli@ cluster05:/home/harli/cluster/tachy-
on/conf
tachyon - env. sh
100% 3525      3.4 KB/s    00:00
[harli@ cluster04 tachyon] $ scp ./conf/tachyon - env. sh
harli@ cluster06:/home/harli/cluster/tachyon/conf
tachyon - env. sh
```

三、启动集群

1. 启动 ZooKeeper

这里选择 cluster05 节点，进入 Kafka 部署目录，输入命令：

```
[harli@ cluster05 Desktop] $ cd ../cluster/kafka/
[harli@ cluster05 kafka] $ vim config/zookeeper. properties
[harli@ cluster05 kafka] $ bin/zookeeper - server - start. sh  - daemon config/zookeeper. properties
[harli@ cluster05 kafka] $ jps
7464 Worker
8030QuorumPeerMain
```

```
8047 Jps
7386DataNode
```

其中，QuorumPeerMain 进程就是启动的 ZooKeeper 进程。

2. 重新格式化 Tachyon

如果是首次修改底层存储系统为 HDFS，则需要进行格式化，具体可以参考第 5.4.1 小节的内容。

3. 启动 Tachyon

```
[harli@ cluster04 tachyon] $ ./bin/tachyon - start. sh allSudoMount
Killed 0 processes
Killed 0 processes
Connection to cluster04 asharli... Killed 0 processes
Connection to cluster04 closed.
Connection to cluster05 asharli... Killed 0 processes
Connection to cluster05 closed.
Connection to cluster06 asharli... Killed 0 processes
Connection to cluster06 closed.
Starting master @ cluster04
Connection to cluster04 asharli... [sudo] password for harli:
FormattingRamFS:/mnt/ramdisk (1gb)
Starting worker @ cluster04
Connection to cluster04 closed.
Connection to cluster05 asharli... [sudo] password for harli:
FormattingRamFS:/mnt/ramdisk (1gb)
Starting worker @ cluster05
Connection to cluster05 closed.
Connection to cluster06 asharli... [sudo] password for harli:
FormattingRamFS:/mnt/ramdisk (1gb)
Starting worker @ cluster06
Connection to cluster06 closed.
```

查看 Web Interface 界面，出现如图 5.18 所示内容表示。

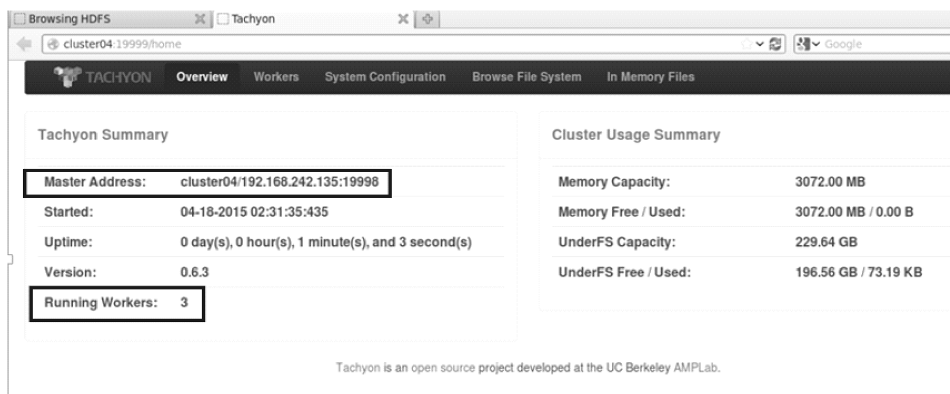


图 5.18 Tachyon 启动后的界面

启动成功。

四、测试 Master 容错性

Tachyon 也是一个 M/S 结构的框架，对应地，也会存在单点故障的问题，这里测试 Tachyon 提供的 Master 节点的单点故障的解决方案。

1. 启动其他 Master 节点

修改配置文件./conf/tachyon－env. sh，这里将 cluster06 作为 Standby Master 节点，输入命令：

```
[harli@ cluster06 tachyon] $ vim ./conf/tachyon－env. sh
```

修改 cluster06 节点上配置文件中的 Tachyon_MASTER_ADDRESS 环境变量，具体如下：

```
export Tachyon_MASTER_ADDRESS = cluster06
```

2. 启动 Master 进程

```
[harli@ cluster06 tachyon] $ ./bin/tachyon－start. sh master
Starting master @ cluster06
[harli@ cluster06 tachyon] $ jps
2374 Worker
2305DataNode
6981TachyonWorker
7046TachyonMaster
7075 Jps
```

启动的 Master 进程 TachyonWorker，就是 Standby Master。

3. 关闭 cluster04 上的 Master 进程

```
[harli@ cluster04 tachyon] $ jps
8344 Worker
16486 Jps
16258TachyonMaster
7837DataNode
7734NameNode
8022SecondaryNameNode
8188 Master
16395TachyonWorker
[harli@ cluster04 tachyon] $ kill －9 16258
```

查看新的 Master 上的 Web Interface（http://cluster06:19999）界面，如图 5.19 所示。

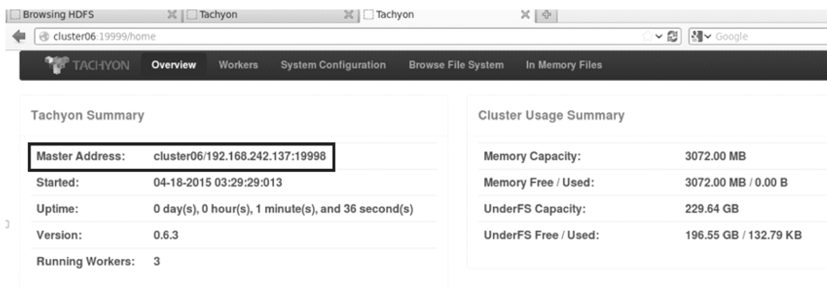


图 5.19　Tachyon 切换 master 后的界面

4. 查看 Active 与 Standby Master 的切换

切换 Master 的时间大概在十几秒左右，从图 5.19 的界面可以看到，Master 节点已经成功切换到 cluster06 了。

需要注意的两点：

1）作为 Master 节点时，需要已经配置 ssh 无密登录到其他节点。

2）另外，当前测试时，需要在希望启动 TachyonMaster 进程的这个节点上，修改属性配置，然后在这个节点上启动 TachyonMaster 进程。修改属性配置 Tachyon_MASTER_ADDRESS 为其他节点的地址时，比如修改为 cluster05，然后调用脚本 ./bin/tachyon - start.sh master 启动 TachyonMoster 进程的话，这个修改后的属性 TACHYON MASTER ADDRESS 的值对应的其他节点，如 cluster05，是不会启动 TachyonMaster 进程的，查看日志：

```
[harli@cluster06 tachyon]$  vim ./logs/master.out
```

报错信息如下：

```
Exception in thread "main"  java.lang.RuntimeException:tachyon.org.apache.thrift.transport.
TTransportException:Could not create ServerSocket on address cluster05/192.168.242.136:19998.
        at com.google.common.base.Throwables.propagate(Throwables.java:160)
        at tachyon.master.TachyonMaster. <init>(TachyonMaster.java:134)
        at tachyon.master.TachyonMaster.main(TachyonMaster.java:60)
Caused by:tachyon.org.apache.thrift.transport.TTransportException:Could not create ServerSocket on
address cluster05/192.168.242.136:19998.
        at tachyon.org.apache.thrift.transport.TNonblockingServerSocket. <init>(TNonblockingSer-
verSocket.java:89)
        at tachyon.org.apache.thrift.transport.TNonblockingServerSocket. <init>(TNonblockingSer-
verSocket.java:72)
        at tachyon.master.TachyonMaster. <init>(TachyonMaster.java:110)
        ... 1 more
```

五、Tachyon 容错机制的扩展

除了使用高可用（High Availability，HA）方式对 Master 节点进行容错外，对于具体的文件数据，Tachyon 借鉴了 Spark，使用血统关系（Lineage）进行容错。文件元数据中记录了文件之间的依赖关系，当文件丢失时，能够根据依赖关系进行重计算来恢复文件数据。

Section 5.4 Tachyon 配置的案例与解析

以目前常用的 HDFS 底层存储系统进行案例与解析，并在此基础上，进一步提供 Tachyon 各个配置属性的解析。

5.4.1 底层存储系统的配置案例与解析

这一节以目前大数据平台最常用的 HDFS 作为底层存储系统，进行配置案例与解析。

一、环境配置

1. 修改配置文件 conf/tachyon - env. sh，添加或修改下面三个环境变量

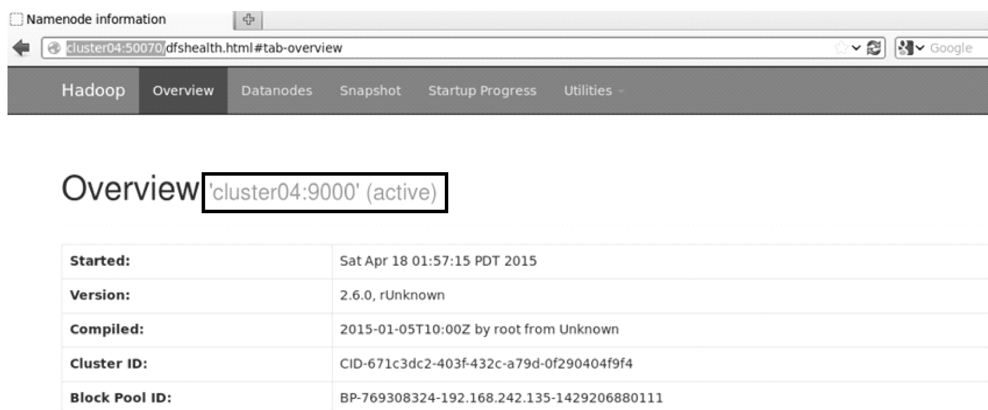
```
JAVA_HOME = "/lib/jdk1.7.0_71"
export Tachyon_MASTER_ADDRESS = cluster04
export Tachyon_UNDERFS_ADDRESS = hdfs://cluster04:9000
```

其中：

1）JAVA_HOME 为本地 Java 安装目录。

2）Tachyon_MASTER_ADDRESS 为 master 对外可见的地址，这里设置为 Master 的 hostname，即 cluster04。

3）Tachyon_UNDERFS_ADDRESS 为底层文件系统，这里使用 HDFS，HDFS 的地址可以查看配置文件，也可以直接从界面查看，Web Interface（http://cluster04:50070/）界面如下，其中带方框的地方就是当前 Hadoop 集群的 HDFS 地址，如图 5.20 所示。



Started:	Sat Apr 18 01:57:15 PDT 2015
Version:	2.6.0, rUnknown
Compiled:	2015-01-05T10:00Z by root from Unknown
Cluster ID:	CID-671c3dc2-403f-432c-a79d-0f290404f9f4
Block Pool ID:	BP-769308324-192.168.242.135-1429206880111

图 5.20　Hadoop 的概览界面

2. 修改配置文件 conf/tachyon - env. sh，添加或修改属性配置

修改前的属性配置：

```
export Tachyon_JAVA_OPTS + = "
  - Dlog4j. configuration = file: $ CONF_DIR/log4j. properties
  - Dtachyon. debug = false
  - Dtachyon. worker. hierarchystore. level. max = 1
  - Dtachyon. worker. hierarchystore. level0. alias = MEM
  - Dtachyon. worker. hierarchystore. level0. dirs. path = $ Tachyon_RAM_FOLDER
 - Dtachyon. worker. hierarchystore. level0. dirs. quota = $ Tachyon_WORKER_MEMORY_SIZE
  - Dtachyon. underfs. address = $ Tachyon_UNDERFS_ADDRESS
  - Dtachyon. underfs. hdfs. impl = $ Tachyon_UNDERFS_HDFS_IMPL
  - Dtachyon. data. folder = $ Tachyon_UNDERFS_ADDRESS/tmp/tachyon/data
  - Dtachyon. workers. folder = $ Tachyon_UNDERFS_ADDRESS/tmp/tachyon/workers
  - Dtachyon. worker. memory. size = $ Tachyon_WORKER_MEMORY_SIZE
  - Dtachyon. worker. data. folder = /tachyonworker/
  - Dtachyon. master. worker. timeout. ms = 60000
  - Dtachyon. master. hostname = $ Tachyon_MASTER_ADDRESS
```

```
-Dtachyon.master.journal.folder=$Tachyon_HOME/journal/
-Dorg.apache.jasper.compiler.disablejsr199=true
-Djava.net.preferIPv4Stack=true
```

修改后的属性配置：



```
export Tachyon_JAVA_OPTS+="
  -Dlog4j.configuration=file:$CONF_DIR/log4j.properties
  -Dtachyon.debug=false
  -Dtachyon.worker.hierarchystore.level.max=1
  -Dtachyon.worker.hierarchystore.level0.alias=MEM
  -Dtachyon.worker.hierarchystore.level0.dirs.path=$Tachyon_RAM_FOLDER
  -Dtachyon.worker.hierarchystore.level0.dirs.quota=$Tachyon_WORKER_MEMORY_SIZE
  -Dtachyon.underfs.address=$Tachyon_UNDERFS_ADDRESS
  -Dtachyon.underfs.hdfs.impl=$Tachyon_UNDERFS_HDFS_IMPL
  -Dtachyon.data.folder=$Tachyon_UNDERFS_ADDRESS/tmp/tachyon/data
  -Dtachyon.workers.folder=$Tachyon_UNDERFS_ADDRESS/tmp/tachyon/workers
  -Dtachyon.worker.memory.size=$Tachyon_WORKER_MEMORY_SIZE
  -Dtachyon.worker.data.folder=/tachyonworker/
  -Dtachyon.master.worker.timeout.ms=60000
  -Dtachyon.master.hostname=$Tachyon_MASTER_ADDRESS
  -Dtachyon.master.journal.folder=$Tachyon_UNDERFS_ADDRESS/journal/
  -Dorg.apache.jasper.compiler.disablejsr199=true
  -Djava.net.preferIPv4Stack=true
```

修改 tachyon.master.journal.folder 配置属性，将 journal 的目录放置于 HDFS 上；

注意：修改完成后，需要将该配置文件复制到集群中的其他节点中，命令如下：

```
[harli@cluster04 tachyon]$ scp ./conf/tachyon-env.sh harli@cluster05:/home/harli/cluster/tachyon/conf
tachyon-env.sh
100% 3525     3.4KB/s    00:00
[harli@cluster04 tachyon]$ scp ./conf/tachyon-env.sh harli@cluster06:/home/harli/cluster/tachyon/conf
tachyon-env.sh
```

3. 重新格式化 Tachyon

由于当前底层文件系统已经修改为 HDFS，因此需要重新进行格式化，构建相关目录，用于存放文件系统的元数据等信息。

```
[harli@cluster04 tachyon]$ vim conf/tachyon-env.sh
[harli@cluster04 tachyon]$ ./bin/tachyon format
Connection to cluster04 asharli... Formatting Tachyon Worker @ cluster04
Connection to cluster04 closed.
Connection to cluster05 asharli... Formatting Tachyon Worker @ cluster05
Connection to cluster05 closed.
Connection to cluster06 asharli... Formatting Tachyon Worker @ cluster06
Connection to cluster06 closed.
Formatting Tachyon Master @ cluster04
[harli@cluster04 tachyon]$
```

4. 验证

查看 HDFS 文件系统信息，进入 Web Interface 界面（http: cluster04: 50070），如图 5.21 所示。

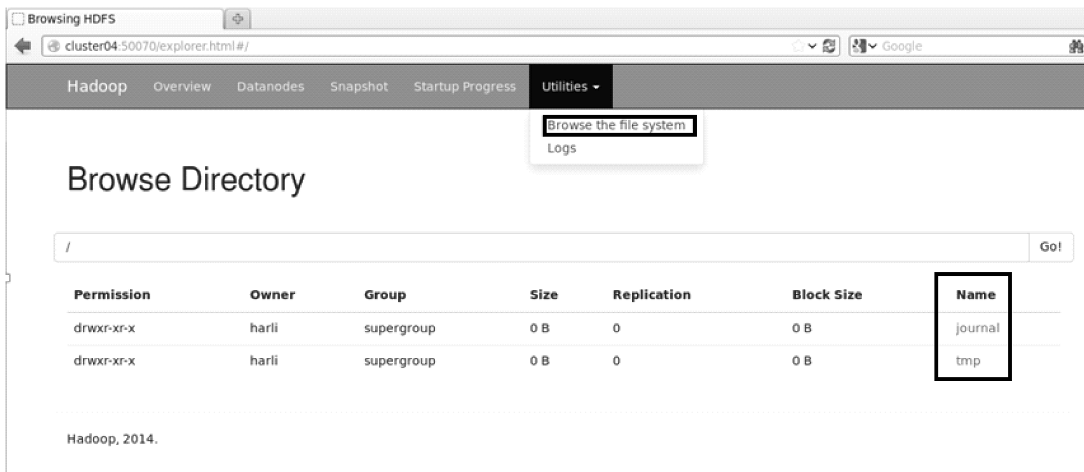


图 5.21　Tachyon 格式化的后底层文件系统信息

可以看到，在 HDFS 的根目录下，已经成功创建了放置 journal 的目录和放置 Tachyon 的文件系统信息的 tachyon. data. folder 和 tachyon. workers. folder 目录。

5.4.2 配置属性与解析

Tachyon 的程序运行参数配置分为四类：Master 配置，Worker 配置，通用配置（Master 和 Worker）和用户配置。可以在配置文件 conf/tachyon – env. sh 中修改或添加这些配置属性。通常在提供的 conf/tachyon – env. sh. template 上进行修改，先复制该文件，然后在 Tachyon_JAVA_OPTS 中定义这些配置属性。另外，Java VM 的参数，对于 Master，可以使用 Tachyon_MASTER_JAVA_OPTS 进行设置，对于 Worker，可以使用 Tachyon_WORKER_JAVA_OPTS 进行设置。比如，如果要远程调试 Master 的话，可以修改 Tachyon_MASTER_JAVA_OPTS，如果以 7001 端口进行远程调试的话，可以设置为：

```
export Tachyon_MASTER_JAVA_OPTS = " $ Tachyon_JAVA_OPTS  – agentlib:jdwp = transport = dt_
socket,server = y,suspend = n,address = 7001"
```

启动 Master 后，会开始监听 7001 端口，直到 IDE 等工具开始调试。

官方网站上的一些属性配置默认值和源码中有差异，且部分属性未列出。可以查看相关类来确认，这里以官方网站为准；另外属性配置文件 tachyon – env. sh. template 中也提供了一些配置属性的默认值。属性配置相关的类在 tachyon. conf 路径下。

一、通用配置

常见的配置用于 Master 和 Worker，包含指定路径的常量和日志 appender 的名字。参见表 5.1。



对应源码类为：tachyon. conf. CommonConf。

表 5.1 通用配置及其含义

配置属性	默认值	含义
tachyon. home	/mnt/tachyon_default_home	Tachyon 的安装路径。这个配置很重要，比如，Master 会对各个 Worker 的状态进行监控，发现 Worker 失效时会自动重启对应的 Worker。而重启时的命令就是在 Tachyon 的安装路径下查找的
tachyon. underfs. address	$ tachyon. home + "/underfs"	Tachyon 在底层文件系统中的路径
tachyon. data. folder	$ tachyon. underfs. address + "/tmp/tachyon/data"	Tachyon 数据在底层文件系统的存放路径
tachyon. workers. folder	$ tachyon. underfs. address + "/tmp/tachyon/workers"	TachyonWorkers 在底层文件系统的存放路径
tachyon. usezookeeper	false	是否使用 ZooKeeper 设置 Master 容错
tachyon. zookeeper. adress	null	Master 容错时 ZooKeeper 的地址，仅 tachyon. usezookeeper 为 true 时有效
tachyon. zookeeper. election. path	/election	Zookeeper 的 election 文件夹，仅 tachyon. usezookeeper 为 true 时有效
tachyon. zookeeper. leader. path	/leader	Zookeeper 的 leader 文件夹路径，仅 tachyon. usezookeeper 为 true 时有效
tachyon. underfs. hdfs. impl	org. apache. hadoop. hdfs. DistributedFileSystem	在使用 HDFS 作为底层存储系统时，所用的 HDFS 实现类。Tachyon 实现 HDFS 的接口，所以可以用和 HDFS 支持的其他文件系统一样的方式去使用，即通过指定 Scheme 来识别文件系统
tachyon. max. columns	1000	Tachyon 中 RawTable 允许的最大列数，必须在 Client 和 Server 侧同时设置
tachyon. table. metadata. byte	5242880，即 5 MB	Tachyon 中 RawTable 元数据允许存储的最大字节数。必须在 Server 侧设置。注：在源码中没有找到该属性。应该改为 tachyon. max. table. metadata. byte 属性
fs. s3n. awsAccessKeyId	null	在将 S3 作为底层存储系统时，S3 的 AWS Access Key Id，是常见的用户名，用来区分用户
fs. s3n. awsSecretAccessKey	null	在将 S3 作为底层存储系统时，S3 的 AWS secretKey Id，是用于计算签名的
tachyon. underfs. glusterfs . mounts	null	使用 GlusterFS 为底层文件系统时，GlusterFS 卷的挂载目录，比如：/vol
tachyon. underfs. glusterfs. volumes	null	使用 GlusterFS 为底层文件系统时，GlusterFS 的卷名，比如：/tachyon_vol
tachyon. underfs. glusterfs. mapred. system. dir	glusterfs:///mapred/system	可选配置，使用 GlusterFS 作为底层文件系统时，GlusterFS 用于存放 MapReduce 中间数据的子目录
tachyon. underfs. hadoop. prefixes	hdfs://s3://s3n://glusterfs:///	可选配置，底层文件系统通过 Hadoop 实现时，使用的前缀列表，分隔符可以是任何空白分隔符和/或","
tachyon. master. retry	29	连接 Master 失败时的重试次数。29 次是最大值，会以指数方式递增重试的时间

二、Master 配置

用于 Master 节点的配置信息，比如 Master 节点使用的地址和端口号等。参见表 5.2。

对应源码类为：tachyon. conf. MasterConf。

表 5.2　Master 配置及其含义

配置属性	默认值	含义
tachyon. master. journal. folder	$ tachyon. home + "/journal/"	存放 Master 的 Journal 日志的路径。Master 中保存的元数据使用 Journal 进行容错，具体包括 Editlog——记录所有对元数据的操作，以及 Image——持久化元数据信息
tachyon. master. hostname	localhost	TachyonMaster 对外可见的主机名
tachyon. master. port	19998	TachyonMaster 的通讯端口
tachyon. master. web. port	19999	TachyonMaster 的 Web Interface 端口
tachyon. master. whitelist	/	可缓存的路径前缀列表，列表以逗号隔开，表示该路径下的文件可以被缓存到内存。当文件可缓存时，第一次读取该文件后就会尝试缓存到内存中。可以通过该属性设置优先缓存的路径，并将使用频繁的文件存放在该路径下，提高性能
tachyon. master. web. threads	1	TachyonMaster 用于 Web Server 的线程数
tachyon. master. keytab. file		用于 Tachyon master 的 Kerberoskeytab 文件
tachyon. master. principal		用于 Tachyon master 的 Kerberos principal

三、Worker 配置

用于 Masters 节点的配置信息，比如 Worker 节点使用的地址和端口号等，参见表 5.3。

对应源码类为：tachyon. conf. WorkerConf，部分属性配置在其他类中（如 tachyon. master 路径下的两个类）。

表 5.3　Worker 配置及其含义

配置属性	默认值	含义
tachyon. work. port	29998	TachyonWorker 的通讯端口
tachyon. worker. data. port	29999	TachyonWorker 的数据传输服务的端口
tachyon. worker. data. folder	/tachyonworker/	Tachyon 的 worker 节点在各个存储目录中存放数据的相对路径
tachyon. worker. memory. size	128 M	每个 TachyonWorker 节点可使用的内存大小
tachyon. worker. hierarchystore. level. max	1	分层存储中的最大级别。目前存储级别包括 MEM（1）、SSD（2）、HDD（3）三层
tachyon. worker. hierarchystore. level0. alias	MEM	存储最高层的别名。
tachyon. worker. hierarchystore. level0. dirs. path	/mnt/ramdisk/	最高层存储对应的存储路径。注意，macs 中的路径应该使用"/Volumes/"
tachyon. worker. hierarchystore. level0. dirs. quota	$ {tachyon. worker. memory. size}	存储最高层分配的内存大小

（续）

配置属性	默认值	含义
tachyon. worker. allocate. strategy	MAX_FREE	Worker 在特定存储层对存储目录的空间分配策略。具体包括：MAX_FREE，在最大剩余空间的存储目录上分配；MAX_FREE，随机进行分配；ROUND_ROBIN，采用圆桌算法分配
tachyon. worker. evict. strategy	LRU	Worker 在存储层空间不足时，使用的块文件释放策略。具体包括：LRU，多个存储目录上的最近最少使用的策略；PARTIAL_LRU，特定最近最少使用的策略
tachyon. worker. network. type	NETTY	Worker 上不同的网络实现，有效值包括 NETTY 和 NIO
tachyon. worker. network. netty. channel	EPOLL	选择 Netty 的 Channel 实现。在 Linux 上使用 epoll。有效值包括 NIO 和 EPOLL
tachyon. worker. network. netty. boss. threads	1	Netty 接受新请求的线程数
tachyon. worker. network. netty. worker. threads	0	Netty 处理请求的线程数。应该设置在默认值 0 和# cpuCores * 2 之间
tachyon. worker. network. netty. file. transfer	MAPPED	向用户返回文件时，所使用的传输方式。有效值包括 MAPPED（使用 java MappedByteBuffer） 和 TRANSFER（使用 Java FileChannel. transferTo）
tachyon. worker. network. netty. watermark. high	32768，即 32 KB	Netty 中 ChannelOutboundBuffer 的高水位线。当 Buffer 的大小超过高水位线的时候对应 Channel 的 isWritable 就会变成 false，当 Buffer 的大小低于低水位线的时候，isWritable 就会变成 true
tachyon. worker. network. netty. watermark. low	8192，即 8 KB	Netty 中 ChannelOutboundBuffer 的低水位线。一旦到达高水位线，在切换到可写状态前，队列就应该刷新到低水位线
tachyon. worker. network. netty. backlog	128 on linux	在拒绝新的请求之前，队列可以缓存的请求数。该值依赖于具体的平台
tachyon. worker. network. netty. buffer. send	平台相关	为 Socket 设置 SO_SNDBUF。更多细节可以在 Socket 帮助页查看
tachyon. worker. network. netty. buffer. receive	平台相关	为 Socket 设置 SO_RCVBUF。更多细节可以在 Socket 帮助页查看
tachyon. worker. keytab. file		Tachyon Worker 的 Kerberoskeytab 文件
tachyon. worker. principal		Tachyon Worker 的 Kerberos principal

四、用户配置

用户配置信息，指定文件系统访问相关的属性值。参见表 5. 4。

对应源码类为：tachyon. conf. UserConf。

表 5.4　用户配置及其含义

配置属性	默认值	含义
tachyon. user. failed. space. request. limits	3	向文件系统请求空间失败前的最大重试次数
tachyon. user. file. writetype. default	CACHE_THROUGH	默认 Tachyon 文件的写类型，包括命令行接口（CLI）中的 copyFromLocal 和 Hadoop 兼容接口。可以是 WriteType 定义的任何类型
tachyon. user. quota. unit. bytes	8 MB	用户一次向 Tachyon Worker 请求的最少字节数
tachyon. user. file. buffer. byte	1 MB	文件系统读写文件时的缓存大小
tachyon. user. default. block. size. byte	1 GB	Tachyon 文件的默认块大小。源码中默认值为 512 MB
tachyon. user. remote. read. buffer. size. byte	8 MB	用户从远程 Tachyon Worker 读取文件时的缓冲大小
tachyon. worker. network. netty. process. threads	16	用于处理块请求的线程数。注：该属性应该属于 Worker 配置，且默认值为 0，源码中位于 WorkerConf 类

Section 5.5 命令行接口的案例与解析

进行命令行接口的案例实践时，已经部署了 Tachyon 集群，并且配置了容错处理，即启动了 ZooKeeper 对 Master 进行管理；启动了 HDFS，同时将 HDFS 配置为 Tachyon 的底层文件系统。

Tachyon 的命令行界面让用户可以对文件系统进行基本的操作。调用命令行工具使用以下脚本：

```
./bin/tachyontfs
```

文件系统访问的路径格式如下：

```
tachyon://<master node address>:<master node port>/<path>
```

其中 tachyon://<master node address>:<master node port>前缀可以省略，可以从配置文件中读取。

注意：如果使用路径的前缀，需要指定 Active 的 Master 节点地址，比如，使用地址为：192.168.242.137:19998。该地址信息可以从 Web Interface 界面获取。

5.5.1 命令行接口的说明

可以通过 Tachyon 的 tfs 的 help 选项，查看命令接口的说明信息。具体命令如下所示：

```
[harli@cluster04 tachyon]$ ./bin/tachyontfs -help
```



```
Usage:javaTFsShell
        [cat <path>]
        [count <path>]
        [ls <path>]
        [lsr <path>]
        [mkdir <path>]
        [rm <path>]
        [rmr <path>]
        [tail <path>]
        [touch <path>]
        [mv <src> <dst>]
        [copyFromLocal <src> <remoteDst>]
        [copyToLocal <src> <localDst>]
        [fileinfo <path>]
        [location <path>]
        [report <path>]
        [request <tachyonaddress> <dependencyId>]
        [pin <path>]
        [unpin <path>]
        [free <file path|folder path>]
```

其中大部分的命令含义可以参考 Linux 下同名命令，命令及其含义参见表 5.5。

表 5.5　Tachyon 命令行接口的命令及其含义

命　令	含　义
cat	将文件内容输出到控制台
count	显示匹配指定的前缀“路径”的文件夹和文件的数量
ls	列出指定路径下所有的文件和目录信息，如大小等
lsr	递归地列出指定路径下所有的文件和目录信息，如大小等
mkdir	在给定的路径创建一个目录，以及任何必要的父目录。如果路径已经存在将会失败
rm	删除一个文件。如果是一个目录的路径将会失败
rmr	删除一个文件或目录，以及该目录下的所有文件夹和文件
tail	输出指定文件的最后 1 kb 到控制台
touch	在指定的路径创建一个 0 字节的文件
mv	移动指定的源文件或源目录到一个目的路径。如果目的路径已经存在将会失败
copyFromLocal	将本地指定的路径复制到 Tachyon 中指定的路径。如果 Tachyon 中指定的路径已经存在将会失败
copyToLocal	从 Tachyon 中指定的路径复制本地指定的路径
fileinfo	输出指定文件的块信息
location	输出存放指定文件的所在节点列表信息
report	向 Master 报告文件丢失
request	根据指定的 dependency ID，请求文件
pin	将指定的路径常驻在内存中。如果指定的是一个文件夹，会递归地包含所有文件以及任何在这个文件夹中新创建的文件

（续）

命　　令	含　　义
unpin	撤销指定路径的常驻内存状态。如果指定的是一个文件夹，会递归地包含所有文件以及任何在这个文件夹中新创建的文件
free	释放一个文件或一个文件夹下的所有文件的内存。文件/文件夹在 underfs（底层文件系统）仍然是可用的

下面举例一些命令的案例实践及解析。

5.5.2 命令行接口的案例实践与解析

和 Hadoop 一样，Tachyon 也提供了一批命令行接口来对 Tachyaon 文件系统进行操作。本节针对这些命令行接口给出具体的操作案例及其解析。

1. copyFromLocal

首先将本地 conf 目录复制到 Tachyon 文件系统的根目录下的 conf 子目录。

```
[harli@cluster04 tachyon]$ ./bin/tachyontfscopyFromLocal ./conf/conf
Copied ./conf to/conf
```

2. copyToLocal

```
[harli@cluster04 tachyon]$ ./bin/tachyon tfs copyToLocal /conf /home/harli/cluster/tachyon/conflo-
cal
Exception in thread "main" java.lang.NullPointerException
      at tachyon.client.TachyonFile.getInStream(TachyonFile.java:165)
      at tachyon.command.TFsShell.copyToLocal(TFsShell.java:204)
      at tachyon.command.TFsShell.run(TFsShell.java:606)
      at tachyon.command.TFsShell.main(TFsShell.java:58)
[harli@cluster04 tachyon]$ ./bin/tachyon tfs copyToLocal /conf/tachyon-env.sh /home/harli/clus-
ter/tachyon/conflocal
Copied /conf/tachyon-env.sh to /home/harli/cluster/tachyon/conflocal
```

注意：命令中的 src 必须是 Tachyon 文件系统中的文件，不支持目录复制。

3. ls 和 lsr

使用 ls 和 lsr 命令查看 Tachyon 文件系统下的文件信息。其中 lsr 命令可以递归地查看子目录。这里给出了带路径前缀和使用默认前缀的两种 path（即对应有无前缀：tachyon://cluster06:19998）。

```
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls /
0.00 B     04-18-2015 03:46:41:184                    /conf
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls tachyon://cluster06:19998/
0.00 B     04-18-2015 03:46:41:184                    /conf
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls /conf
3525.00 B 04-18-2015 03:46:41:280   In Memory      /conf/tachyon-env.sh
3042.00 B 04-18-2015 03:46:44:857   In Memory      /conf/tachyon-glusterfs
-env.sh.template
83.00 B   04-18-2015 03:46:44:989   In Memory      /conf/slaves
```

```
3356.00 B 04-18-2015 03:46:45:185   In Memory     /conf/tachyon-env.sh.template
114.00 B  04-18-2015 03:46:45:309   In Memory     /conf/workers
1971.00 B 04-18-2015 03:46:45:433   In Memory     /conf/log4j.properties

[harli@cluster04 tachyon]$ ./bin/tachyon tfs lsr /
0.00 B    04-18-2015 03:46:41:184                 /conf
3525.00 B 04-18-2015 03:46:41:280   In Memory     /conf/tachyon-env.sh
3042.00 B 04-18-2015 03:46:44:857   In Memory     /conf/tachyon-glusterfs
-env.sh.template
83.00 B   04-18-2015 03:46:44:989   In Memory     /conf/slaves
3356.00 B 04-18-2015 03:46:45:185   In Memory     /conf/tachyon-env.sh.template
114.00 B  04-18-2015 03:46:45:309   In Memory     /conf/workers
1971.00 B 04-18-2015 03:46:45:433   In Memory     /conf/log4j.properties
```



4. count

统计当前路径下的目录、文件信息，包括文件数、目录树以及总的大小。

```
[harli@cluster04 tachyon]$ ./bin/tachyon tfs count /
File Count               Folder Count              Total Bytes
6                          2                             12091
[harli@cluster04 tachyon]$ ./bin/tachyon tfs count /conf
File Count               Folder Count              Total Bytes
6                          1                             12091
```

5. cat

查看指定文件的内容。

```
[harli@cluster04 tachyon]$ ./bin/tachyon tfs cat /conf
/conf is not a file.
[harli@cluster04 tachyon]$ ./bin/tachyon tfs cat /conf/tachyon-env.sh
#! /usr/bin/env bash

# This file contains environment variables required to run Tachyon. Copy it as tachyon-env.sh and
#edit that to configure Tachyon for your site. At a minimum,
# the following variables should be set:
……
```

6. mkdir、rm、rmr 和 touch

其中：

1）mkdir：创建目录，支持自动创建不存在的父目录。

2）rm：删除文件，不能删除目录。

3）rmr：删除目录，支持递归，包含子目录和文件。注意，递归删除根目录是无效的。

4）touch：创建文件，不能创建已经存在的文件。

```
[harli@cluster04 tachyon]$ ./bin/tachyon tfs mkdir /mydir
Successfully created directory /mydir
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls /
0.00 B    04-18-2015 03:46:41:184                 /conf
0.00 B    04-18-2015 04:08:01:730                 /mydir
```

```
[harli@ cluster04 tachyon] $ ./bin/tachyon tfs touch /mydir2/2/2/my. txt
/mydir2/2/2/my. txt has been created
[harli@ cluster04 tachyon] $ ./bin/tachyon tfs lsr  /mydir2
0.00 B     04 -18 -2015 04:15:25:052                  /mydir2/2
0.00 B     04 -18 -2015 04:15:25:052                  /mydir2/2/2
0.00 B     04 -18 -2015 04:15:25:052   In Memory      /mydir2/2/2/my. txt

[harli@ cluster04 tachyon] $ ./bin/tachyon tfs rm /mydir
can't remove a directory,please tryrmr <path>
[harli@ cluster04 tachyon] $ ./bin/tachyontfs touch /mydir/my. txt
/mydir/my. txt has been created

[harli@ cluster04 tachyon] $ ./bin/tachyontfs touch /mydir/my. txt
FileAlreadyExistException(message:/mydir/my. txt)
[harli@ cluster04 tachyon] $ ./bin/tachyontfs ls /mydir
0.00 B     04 -18 -2015 04:09:31:670   In Memory      /mydir/my. txt

[harli@ cluster04 tachyon] $ ./bin/tachyontfs rm /mydir/my. txt
/mydir/my. txt has been removed

[harli@ cluster04 tachyon] $ ./bin/tachyontfsrmr  /mydir2
/mydir2 has been removed

[harli@ cluster04 tachyon] $ ./bin/tachyontfsrmr /
[harli@ cluster04 tachyon] $ ./bin/tachyontfs ls /
0.00 B     04 -18 -2015 03:46:41:184                  /conf
0.00 B     04 -18 -2015 04:08:01:730                  /mydir
0.00 B     04 -18 -2015 04:14:49:422                  /mydir1
```

7. location

查看文件所在位置，当前/conf/tachyon - env. sh 文件位于 cluster04 节点。

```
[harli@ cluster04 tachyon] $ ./bin/tachyontfs location /conf
Exception in thread "main" java. lang. NullPointerException
     at tachyon. client. TachyonFile. getNumberOfBlocks(TachyonFile. java:222)
     at tachyon. client. TachyonFile. getLocationHosts(TachyonFile. java:203)
     at tachyon. command. TFsShell. location(TFsShell. java:314)
     at tachyon. command. TFsShell. run(TFsShell. java:610)
     at tachyon. command. TFsShell. main(TFsShell. java:58)
[harli@ cluster04 tachyon] $ ./bin/tachyontfs location /conf/tachyon - env. sh
/conf/tachyon - env. sh with file id 3 is on nodes:
cluster04
```

8. report

report 会报告指定路径的具体信息，包含对应的文件 ID 标识的值以及已经上报的信息等。

```
[harli@ cluster04 tachyon] $ ./bin/tachyontfs report /conf/tachyon - env. sh
```

```
/conf/tachyon-env.sh with file id 3 has reported been report lost.
[harli@cluster04 tachyon]$ ./bin/tachyontfs report /conf
/conf with file id 2 has reported been report lost.
[harli@cluster04 tachyon]$ ./bin/tachyontfs report /mydir2
/mydir2 with file id -1 has reported been report lost.
```

9. request

向指定文件发出请求。

```
[harli@cluster04 tachyon]$ ./bin/tachyontfs request /conf 8
Dependency with ID 8 has been requested.
[harli@cluster04 tachyon]$ ./bin/tachyontfs request /conf 7
Dependency with ID 7 has been requested.
```

10. pin 和 unpin

对应常驻内存和撤销常驻内存。

```
[harli@cluster04 tachyon]$ ./bin/tachyontfs pin /conf/log4j.properties
File '/conf/log4j.properties' was successfully pinned.
[harli@cluster04 tachyon]$ ./bin/tachyontfs unpin /conf/log4j.properties
File '/conf/log4j.properties' was successfully unpinned.
```

pin 执行前或 unpin 执行后的 Web Interface 界面，如图 5.22 所示。

File Name	Size	Block Size	In-Memory	Pin	Creation Time	Modification Time
log4j.properties	1971.00 B	512.00 MB	100%	NO	04-18-2015 03:46:45:433	04-18-2015 03:46:45:558
slaves	83.00 B	512.00 MB	100%	NO	04-18-2015 03:46:44:989	04-18-2015 03:46:45:182
tachyon-env.sh	3525.00 B	512.00 MB	100%	NO	04-18-2015 03:46:41:280	04-18-2015 03:46:44:852
tachyon-env.sh.template	3356.00 B	512.00 MB	100%	NO	04-18-2015 03:46:45:185	04-18-2015 03:46:45:306
tachyon-glusterfs-env.sh.template	3042.00 B	512.00 MB	100%	NO	04-18-2015 03:46:44:857	04-18-2015 03:46:44:985
test				NO	04-18-2015 05:16:37:529	04-18-2015 05:16:37:529

图 5.22 Tachyon 文件系统 pin 或 unpin 执行前的界面

pin 执行后的 Web Interface 界面，如图 5.23 所示。

File Name	Size	Block Size	In-Memory	Pin	Creation Time	Modification Time
log4j.properties	1971.00 B	512.00 MB	100%	YES	04-18-2015 03:46:45:433	04-18-2015 05:22:11:383
slaves	83.00 B	512.00 MB	100%	NO	04-18-2015 03:46:44:989	04-18-2015 03:46:45:182
tachyon-env.sh	3525.00 B	512.00 MB	100%	NO	04-18-2015 03:46:41:280	04-18-2015 03:46:44:852
tachyon-env.sh.template	3356.00 B	512.00 MB	100%	NO	04-18-2015 03:46:45:185	04-18-2015 03:46:45:306
tachyon-glusterfs-env.sh.template	3042.00 B	512.00 MB	100%	NO	04-18-2015 03:46:44:857	04-18-2015 03:46:44:985

图 5.23 Tachyon 文件系统 pin 或 unpin 执行后的界面

11. free

释放内存，即文件不再放在内存中。

```
[harli@ cluster04 tachyon] $ ./bin/tachyontfs free /conf/log4j. properties
/conf/log4j. properties was successfully freed from memory.
```

执行该 free 命令后，查看 Web Interface 界面，在 In - Memory 列可以看到文件已经不在内存中，如图 5.24 所示。

File Name	Size	Block Size	In-Memory	Pin	Creation Time	Modification Time
log4j.properties	1971.00 B	512.00 MB	0%	NO	04-18-2015 03:46:45:433	04-18-2015 05:23:07:210
slaves	83.00 B	512.00 MB	100%	NO	04-18-2015 03:46:44:989	04-18-2015 03:46:45:182
tachyon-env.sh	3525.00 B	512.00 MB	100%	NO	04-18-2015 03:46:41:280	04-18-2015 03:46:44:852
tachyon-env.sh.template	3356.00 B	512.00 MB	100%	NO	04-18-2015 03:46:45:185	04-18-2015 03:46:45:306
tachyon-glusterfs-env.sh.template	3042.00 B	512.00 MB	100%	NO	04-18-2015 03:46:44:857	04-18-2015 03:46:44:985

图 5.24　Tachyon 文件系统 free 操作后的界面

Section 5.6 同步底层文件系统的案例与解析

进行同步底层文件系统的案例实践时，要求已经部署了 Tachyon 集群，并且配置了容错处理，即启动了 ZooKeeper 对 Master 进行管理，启动了 HDFS，同时将 HDFS 配置为 Tachyon 的底层文件系统。

Tachyon 启动后并不能识别已经存储在底层文件系统中的数据，可以使用 Tachyon 的 shell 命令 loadufs 来同步底层文件系统和本地文件系统。命令格式如下：

```
./bin/tachyonloadufs [Tachyon_PATH] [UNDERLYING_FILESYSTEM_PATH] [Optional EXCLUDE_
PATHS]
```

其中：

1）Tachyon_PATH：指定要同步的 Tachyon 路径。

2）UNDERLYING_FILESYSTEM_PATH：指定底层文件需要要同步的路径。

3）EXCLUDE_PATHS：可选参数，排除同步路径下不需要同步的路径。

需要准备的文件如下。

```
[harli@ cluster06 tachyon] $ hdfs dfs -mkdir /test
[harli@ cluster06 tachyon] $ hdfs dfs -put ./bin /test
[harli@ cluster06 tachyon] $ hdfs dfs -put ./logs/ /test
[harli@ cluster06 tachyon] $ hdfs dfs -put ./conf /test
[harli@ cluster06 tachyon] $ hdfs dfs -ls /test
Found 3 items
```

```
drwxr-xr-x   - harli supergroup          0 2015-04-18 09:39 /test/bin
drwxr-xr-x   - harli supergroup          0 2015-04-18 09:40 /test/conf
drwxr-xr-x   - harli supergroup          0 2015-04-18 09:40 /test/logs

[harli@cluster06 tachyon]$ hdfs dfs -put ./conf /test/bin
[harli@cluster06 tachyon]$ hdfs dfs -put ./logs /test/bin
[harli@cluster06 tachyon]$ hdfs dfs -ls /test/bin
Found 7 items
drwxr-xr-x   - harli supergroup          0 2015-04-18 09:55 /test/bin/conf
drwxr-xr-x   - harli supergroup          0 2015-04-18 09:56 /test/bin/logs
-rw-r--r--   1harli supergroup       7084 2015-04-18 09:39 /test/bin/tachyon
-rw-r--r--   1harli supergroup       4374 2015-04-18 09:39 /test/bin/tachyon-mount.sh
-rw-r--r--   1harli supergroup       5161 2015-04-18 09:39 /test/bin/tachyon-start.sh
-rw-r--r--   1harli supergroup       1102 2015-04-18 09:39 /test/bin/tachyon-stop.sh
-rw-r--r--   1harli supergroup        715 2015-04-18 09:39 /test/bin/tachyon-workers.sh
```

其中，目录结构为：

```
[harli@cluster06 tachyon]$ hdfs dfs -ls -d -R /test/bin/*
drwxr-xr-x   - harli supergroup          0 2015-04-18 09:55 /test/bin/conf
drwxr-xr-x   - harli supergroup          0 2015-04-18 09:56 /test/bin/logs
-rw-r--r--   1harli supergroup       7084 2015-04-18 09:39 /test/bin/tachyon
-rw-r--r--   1harli supergroup       4374 2015-04-18 09:39 /test/bin/tachyon-
mount.sh
-rw-r--r--   1harli supergroup       5161 2015-04-18 09:39 /test/bin/tachyon-start.sh
-rw-r--r--   1harli supergroup       1102 2015-04-18 09:39 /test/bin/tachyon-stop.sh
-rw-r--r--   1harli supergroup        715 2015-04-18 09:39 /test/bin/tachyon-work-
ers.sh
```

5.6.1 同步 HDFS 底层文件系统的案例与解析

在使用 EXCLUDE_PATHS 选项时，如果包含多项排除路径，需要用双引号包含，否则会将分号视作命令分隔符，继续执行分号后的命令，如下所示：

```
[harli@cluster04 tachyon]$  ./bin/tachyonloadufs tachyon://cluster06:19998/ hdfs://cluster04:
9000/test logs;conf
bash:conf:command not found
[harli@cluster04 tachyon]$ ./bin/tachyontfs ls /
0.00 B    04-18-2015 09:48:21:953         /bin
0.00 B    04-18-2015 09:48:21:982         /conf
```

这里 EXCLUDE_PATHS 选项包含 logs;conf，但由于没有用双引号包含，因此将分号看成了命令分隔符，在继续执行 conf 命令时，会报错。

对应的，只会排除 EXCLUDE PATH 选项中的 logs 目录，其他非 EXCLODE PATHS 选项中的目录都已经同步成功。

同步 HDFS 底层文件系统后，底层文件系统上的同步路径下有文件更新时，并不会自动地同步 HDFS 底层文件系统。

另外 EXCLUDE_PATHS 选项是指定的相对路径，不会递归地去排除，如：

```
[harli@cluster04 tachyon]$ ./bin/tachyon tfs rmr /test
/test has been removed
[harli@cluster04 tachyon]$ ./bin/tachyon tfs mkdir /test
Successfully created directory /test
[harli@cluster04 tachyon]$ ./bin/tachyon loadufs tachyon://cluster06:19998/ hdfs://cluster04:9000/test "logs;conf"
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls /test
[harli@cluster04 tachyon]$ ./bin/tachyon loadufs tachyon://cluster06:19998/test hdfs://cluster04:9000/test "logs;conf"
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls /test
0.00 B     04-18-2015 11:06:04:994                    /test/bin
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls /test/bin
0.00 B     04-18-2015 11:06:05:004                    /test/bin/conf
0.00 B     04-18-2015 11:06:05:038                    /test/bin/logs
6.92 KB    04-18-2015 11:06:05:055   Not In Memory    /test/bin/tachyon
4374.00 B 04-18-2015 11:06:05:075   Not In Memory    /test/bin/tachyon-mount.sh
5.04 KB    04-18-2015 11:06:05:091   Not In Memory    /test/bin/tachyon-start.sh
1102.00 B 04-18-2015 11:06:05:108   Not In Memory    /test/bin/tachyon-stop.sh
715.00 B   04-18-2015 11:06:05:126   Not In Memory    /test/bin/tachyon-workers.sh
```

将 HDFS 的/test 目录同步到 Tachyon 的/test 目录下，并且排除 HDFS 下的/test/logs 和/test/conf 目录，但是，HDFS 下的/test/bin 目录下的 conf 和 logs 并不会排除。

注意： 同步时，会将指定的底层文件系统路径下的数据同步上去，不会把 test 目录也同步上去。

另外，执行命令时，Tachyon 路径必须指定完整路径，不能省略前缀 tachyon://master:19998，否则将包括无效的 Tachyon URI 错误：

```
[harli@cluster04 tachyon]$ ./bin/tachyon loadufs / hdfs://cluster04:9000/test logs;conf
java.io.IOException:Invalid Tachyon URI:/. Use tachyon://host:port/ ,tachyon-ft://host:port/
    at tachyon.client.TachyonFS.get(TachyonFS.java:89)
    at tachyon.util.UfsUtils.loadUfs(UfsUtils.java:71)
    at tachyon.util.UfsUtils.main(UfsUtils.java:185)
Usage:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils <TachyonPath> <UfsPath> [<Optional ExcludePathPrefix,separated by ;>]
Example:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils tachyon://127.0.0.1:19998/a hdfs://localhost:9000/b c
Example:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils tachyon://127.0.0.1:19998/a file:///b c
Example:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils tachyon://127.0.0.1:19998/a /b c
In the TFS,all files under local FS /b will be registered under /a,except for those with prefix c
bash:conf:command not found
```

5.6.2 同步本地底层文件系统的案例与解析

本地底层文件系统的同步和 HDFS 底层文件系统同步类似。如：

```
[harli@ cluster04 tachyon] $ ./bin/tachyon loadufs tachyon://cluster06:19998/test_1 /home/harli/cluster/tachyon/conf
[harli@ cluster04 tachyon] $ ./bin/tachyon tfs lsr /test_1
3525.00 B 04-18-2015 10:14:28:343   Not In Memory   /test_1/tachyon-env.sh
3042.00 B 04-18-2015 10:14:28:357   Not In Memory   /test_1/tachyon-glusterfs-env.sh.template
83.00 B   04-18-2015 10:14:28:367   Not In Memory   /test_1/slaves
3356.00 B 04-18-2015 10:14:28:386   Not In Memory   /test_1/tachyon-env.sh.template
114.00 B  04-18-2015 10:14:28:393   Not In Memory   /test_1/workers
1971.00 B 04-18-2015 10:14:28:400   Not In Memory   /test_1/log4j.properties
```

EXCLUDE_PATHS 选项用法也是一样的，比如：

```
[harli@ cluster04 tachyon] $ cp -r conf test_2/
[harli@ cluster04 tachyon] $ cp -r  logs test_2
[harli@ cluster04 tachyon] $ cp -r  bin test_2/
[harli@ cluster04 tachyon] $ cp -r  conf test_2/bin/
[harli@ cluster04 tachyon] $ cp -r  logs test_2/bin/
```

构建本地目录 test_2，然后测试同步中 EXCLUDE_PATHS 选项的作用：

```
[harli@ cluster04 tachyon] $ ./bin/tachyon loadufs tachyon://cluster06:19998/test_2 /home/harli/cluster/tachyon/test_2 "logs;conf"
[harli@ cluster04 tachyon] $ ./bin/tachyon tfs ls /test_2
0.00 B    04-18-2015 11:14:52:567                   /test_2/bin
715.00 B  04-18-2015 11:17:30:138   Not In Memory   /test_2/tachyon-workers.sh
6.92 KB   04-18-2015 11:17:30:146   Not In Memory   /test_2/tachyon
1102.00 B 04-18-2015 11:17:30:150   Not In Memory   /test_2/tachyon-stop.sh
5.04 KB   04-18-2015 11:17:30:155   Not In Memory   /test_2/tachyon-start.sh
4374.00 B 04-18-2015 11:17:30:159   Not In Memory   /test_2/tachyon-mount.sh
[harli@ cluster04 tachyon] $ ./bin/tachyon tfs ls /test_2
0.00 B    04-18-2015 11:14:52:567                   /test_2/bin
715.00 B  04-18-2015 11:17:30:138   Not In Memory   /test_2/tachyon-workers.sh
6.92 KB   04-18-2015 11:17:30:146   Not In Memory   /test_2/tachyon
1102.00 B 04-18-2015 11:17:30:150   Not In Memory   /test_2/tachyon-stop.sh
5.04 KB   04-18-2015 11:17:30:155   Not In Memory   /test_2/tachyon-start.sh
4374.00 B 04-18-2015 11:17:30:159   Not In Memory   /test_2/tachyon-mount.sh
```

可以看到，效果和 HDFS 底层文件系统的同步是一样的。

本地文件必须使用“file:///”或“/”作为前缀，不能使用本地的相对路径：

```
[harli@ cluster04 tachyon] $ ./bin/tachyon loadufs tachyon://cluster06:19998/test_1 ./conf
java.lang.NullPointerException
at tachyon.util.UfsUtils.loadUnderFs(UfsUtils.java:102)
at tachyon.util.UfsUtils.loadUfs(UfsUtils.java:75)
at tachyon.util.UfsUtils.main(UfsUtils.java:185)
Usage:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils <TachyonPath> <UfsPath> [<Optional ExcludePathPrefix,separated by ;>]
Example:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils tachyon://127.0.0.1:19998/a hdfs://localhost:9000/b c
```

```
Example:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils tachyon://127.0.0.1:19998/a file:///b c
Example:java -cp target/tachyon-0.6.3-jar-with-dependencies.jar tachyon.util.UfsUtils tachyon://127.0.0.1:19998/a /b c
In the TFS,all files under local FS /b will be registered under /a,except for those with prefix c
```

Section 5.7 基于 Tachyon 运行的案例与解析

5.7.1 基于 Tachyon 运行 Spark 的案例与解析

基于 Tachyon 运行 Spark 的案例实践时，要求已经部署了 Tachyon 集群，并且配置了容错处理，即启动了 ZooKeeper 对 Master 进行管理，启动了 HDFS，同时将 HDFS 配置为 Tachyon 的底层文件系统。

一、准备在 Spark 上使用的文件

```
[harli@cluster04 tachyon]$ ./bin/tachyon tfs copyFromLocal ./conf/tachyon-env.sh /
Copied ./conf/tachyon-env.sh to /
[harli@cluster04 tachyon]$ ./bin/tachyon tfs ls /
3525.00 B 04-19-2015 05:24:17:689   In Memory        /tachyon-env.sh
[harli@cluster04 tachyon]$
```

二、基于 Tachyon 的 Spark 实践案例及解析

1. 修改配置 Spark-env.sh，添加以下内容：

```
export Spark_CLASSPATH=../tachyon/client/target/tachyon-client-0.6.3-jar-with-dependencies.jar:$Spark_CLASSPATH
```

注意：在重新编译的 Spark 1.3.0（不低于 1.0.0）版本中，没有设置也可以成功访问 Tachyon。

2. 增加配置文件 core-site.xml

```
<configuration>
<property>
<name>fs.tachyon.impl</name>
<value>tachyon.hadoop.TFS</value>
</property>
</configuration>
```

注意：在重新编译的 spark 1.3.0 版本（基于 Hadoop 2.x 版本）中，没有设置也可以成功访问 Tachyon。

3. 启动./bin/spark-shell

进入 spark 部署目录，输入：

```
[harli@cluster04 tachyon]$ ./bin/spark-shell
```

去除过多的日志信息：

```
scala > import org. apache. log4j. {Level,Logger}
import org. apache. log4j. {Level,Logger}

scala > Logger. getLogger("org. apache. spark"). setLevel(Level. WARN)
```



4. 使用 Tachyon 文件

启动 ./bin/spark - shell，在交互式界面输入：

```
scala >   val s = sc. textFile("tachyon://cluster04: 19998/tachyon - env. sh")
s:org. apache. spark. rdd. RDD[String] = tachyon://cluster04: 19998/tachyon - env. sh MapPartitionsRDD[1] at textFile at <console>:22
scala >   s. count
15/04/19 08:15:10INFO:initialize(tachyon://cluster04:19998/tachyon - env. sh,Configuration:core - default. xml,core - site. xml, mapred - default. xml, mapred - site. xml, yarn - default. xml, yarn - site. xml,hdfs - default. xml,hdfs - site. xml). Connecting to Tachyon:tachyon://cluster04:19998/tachyon - env. sh
15/04/19 08:15:10WARN:tachyon. home is not set. Using /mnt/tachyon_default_home as the default value.
15/04/19 08:15:10 INFO:Tachyon client (version 0. 6. 3) is trying to connect master @ cluster04/192. 168. 242. 135:19998
15/04/19 08:15:10 INFO: User registered at the master cluster04/192. 168. 242. 135:19998 got UserId 20
15/04/19 08:15:10 INFO: tachyon://cluster04:19998 tachyon://cluster04:19998hdfs://cluster04:9000
15/04/19 08:15:10 INFO:getFileStatus(tachyon://cluster04:19998/tachyon - env. sh):HDFS Path:hdfs://cluster04:9000/tachyon - env. sh TPath:tachyon://cluster04:19998/tachyon - env. sh
15/04/19 08:15:10 INFOFileInputFormat:Total input paths to process:1
15/04/19 08:15:10 INFO deprecation:mapred. tip. id is deprecated. Instead,use mapreduce. task. id
15/04/19 08:15:10 INFO deprecation:mapred. task. id is deprecated. Instead,use mapreduce. task. attempt. id
15/04/19 08:15:10 INFO deprecation:mapred. task. is. map is deprecated. Instead,use mapreduce. task. ismap
15/04/19 08:15:10 INFO deprecation:mapred. task. partition is deprecated. Instead,use mapreduce. task. partition
15/04/19 08:15:10 INFO deprecation:mapred. job. id is deprecated. Instead,use mapreduce. job. id
15/04/19 08:15:10 INFO:open(tachyon://cluster04:19998/tachyon - env. sh,65536)
15/04/19 08:15:10 INFO:Trying to get local worker host:cluster04
15/04/19 08:15:10 INFO:Connecting local worker @ cluster04/192. 168. 242. 135:29998
res1:Long = 86
```

5. 将文件重新保存到 Tachyon 上

```
scala >   s. saveAsTextFile("tachyon://cluster04:19998/save1")
15/04/19 08:15:39 INFO:getWorkingDirectory:/
15/04/19 08:15:39 INFO:getWorkingDirectory:/
```

15/04/19 08:15:39 INFO:getFileStatus(tachyon://cluster04:19998/save1):HDFS Path:hdfs://cluster04:9000/save1 TPath:tachyon://cluster04:19998/save1
15/04/19 08:15:40 INFO:File does not exist:tachyon://cluster04:19998/save1
15/04/19 08:15:40 INFO:getWorkingDirectory:/
15/04/19 08:15:40 INFO:mkdirs(tachyon://cluster04:19998/save1/_temporary/0,rwxrwxrwx)
15/04/19 08:15:41 INFO:open(tachyon://cluster04:19998/tachyon-env.sh,65536)
15/04/19 08:15:41 INFO:getWorkingDirectory:/
15/04/19 08:15:41 INFO:create(tachyon://cluster04:19998/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1/part-00000,rw-r--r--,true,65536,1,536870912,org.apache.hadoop.mapred.Reporter$1@40025295)
15/04/19 08:15:41INFO:Folder /mnt/ramdisk/tachyonworker/users/20 was created!
15/04/19 08:15:41INFO:/mnt/ramdisk/tachyonworker/users/20/23622320128 was created!
15/04/19 08:15:41 INFO:getFileStatus(tachyon://cluster04:19998/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1 TPath:tachyon://cluster04:19998/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1
15/04/19 08:15:41 INFO:getFileStatus(tachyon://cluster04:19998/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1 TPath:tachyon://cluster04:19998/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1
15/04/19 08:15:41 INFO:getFileStatus(tachyon://cluster04:19998/save1/_temporary/0/task_201504190815_0001_m_000000):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0/task_201504190815_0001_m_000000 TPath:tachyon://cluster04:19998/save1/_temporary/0/task_201504190815_0001_m_000000
15/04/19 08:15:41 INFO:File does not exist:tachyon://cluster04:19998/save1/_temporary/0/task_201504190815_0001_m_000000
15/04/19 08:15:41 INFO:rename(tachyon://cluster04:19998/save1/_temporary/0/_temporary/attempt_201504190815_0001_m_000000_1,tachyon://cluster04:19998/save1/_temporary/0/task_201504190815_0001_m_000000)
15/04/19 08:15:41 INFOFileOutputCommitter:Saved output of task' attempt_201504190815_0001_m_000000_1' to tachyon://cluster04:19998/save1/_temporary/0/task_201504190815_0001_m_000000
15/04/19 08:15:41 INFO:listStatus(tachyon://cluster04:19998/save1/_temporary/0):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0
15/04/19 08:15:41 INFO:getFileStatus(tachyon://cluster04:19998/save1):HDFS Path:hdfs://cluster04:9000/save1 TPath:tachyon://cluster04:19998/save1
15/04/19 08:15:41 INFO:getFileStatus(tachyon://cluster04:19998/save1):HDFS Path:hdfs://cluster04:9000/save1 TPath:tachyon://cluster04:19998/save1
15/04/19 08:15:41 INFO:listStatus(tachyon://cluster04:19998/save1/_temporary/0/task_201504190815_0001_m_000000):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0/task_201504190815_0001_m_000000
15/04/19 08:15:41 INFO:getFileStatus(tachyon://cluster04:19998/save1/part-00000):HDFS Path:hdfs://cluster04:9000/save1/part-00000 TPath:tachyon://cluster04:19998/save1/part-00000
15/04/19 08:15:41 INFO:File does not exist:tachyon://cluster04:19998/save1/part-00000
15/04/19 08:15:41 INFO:rename(tachyon://cluster04:19998/save1/_temporary/0/task_201504190815_0001_m_000000/part-00000,tachyon://cluster04:19998/save1/part-00000)

```
15/04/19 08:15:41 INFO:delete(tachyon://cluster04:19998/save1/_temporary,true)
15/04/19 08:15:41 INFO:create(tachyon://cluster04:19998/save1/_SUCCESS,rw-r--r--,true,
65536,1,536870912,null)
```

查看 Tachyon 的 Web Interface（http://cluster04:19999）界面，如图 5.25 所示。

Tachyon | Spark Master at spark://clust... | Namenode information

cluster04:19999/memory?offset=0&limit=6

TACHYON Overview Workers System Configuration Browse File System In Memory Files

File Path	Size	Block Size	Pin	Creation Time	Modification Time
/my_tachyon-env.sh/_SUCCESS	0.00 B	512.00 MB	NO	04-19-2015 05:50:00:411	04-19-2015 05:50:00:456
/my_tachyon-env.sh/part-00000	3525.00 B	512.00 MB	NO	04-19-2015 05:49:59:766	04-19-2015 05:50:00:255
/save1/_SUCCESS	0.00 B	512.00 MB	NO	04-19-2015 08:15:41:463	04-19-2015 08:15:41:510
/save1/part-00000	3525.00 B	512.00 MB	NO	04-19-2015 08:15:41:081	04-19-2015 08:15:41:401

图 5.25　保存文件到 Tachyon 文件系统后的界面

可以看到，文件已经成功保存到 Tachyon 文件系统，目录为保存时的参数 save1。

6. 对 RDD 进行缓存

缓存 RDD 时，对应的 API 没有指定 Tachyon 的文件路径，这时候需要提供配置参数，这里提供了 spark.tachyonStore.url 和 spark.tachyonStore.baseDir，通过这两个配置信息，可以确定 RDD 缓存在 Tachyon 文件系统上的路径。

修改配置文件 spark-default.conf，添加以下内容：

```
spark.tachyonStore.url        tachyon://cluster04:19998
spark.tachyonStore.baseDir /spark
```

spark.tachyonStore.url 是 Spark 连接 Tachyon 用的，如果没有设置，会使用默认 URL（localhost/127.0.0.1:19998）去连接，导致连接失败，比如触发缓存后的错误信息如下：

```
scala>   t.count
15/04/19 08:32:45 INFOFileInputFormat:Total input paths to process:1
15/04/19 08:33:00 INFO:Tachyon client (version 0.6.3) is trying to connect master @ localhost/
127.0.0.1:19998
15/04/19 08:33:10 ERROR:Failed to connect (0) to master localhost/127.0.0.1:19998:
java.net.ConnectException:Connection refused
15/04/19 08:33:10 INFO:Tachyon client (version 0.6.3) is trying to connect master @ localhost/
127.0.0.1:19998
15/04/19 08:33:10 ERROR:Failed to connect (1) to master localhost/127.0.0.1:19998:
java.net.ConnectException:Connection refused
15/04/19 08:33:10 INFO:Tachyon client (version 0.6.3) is trying to connect master @ localhost/
127.0.0.1:19998
```

spark.tachyonStore.baseDir 配置属性为 RDD 缓存在 Tachyon 文件系统中的根目录，如

果没有设置，默认值为 tmp_spark_tachyon，缓存后的 RDD 路径如：/tmp_spark_tachyon/spark - e2fddc0b - 8eb8 - 430d - b0c8 - 0f045f4d23af//spark - tachyon - 20150419083901 - ccd6/15/rdd_1_0。

```
scala >  val t = sc. textFile( "file:///home/harli/cluster/tachyon/conf/log4j. properties" )
t: org. apache. spark. rdd. RDD [ String ]  =  file:///home/harli/cluster/tachyon/conf/log4j. properties
MapPartitionsRDD[5] at textFile at  < console > :22
scala >    import org. apache. spark. storage. StorageLevel
import org. apache. spark. storage. StorageLevel
scala >    t. persist( StorageLevel. OFF_HEAP)
res5:t. type = file:///home/harli/cluster/tachyon/conf/log4j. properties MapPartitionsRDD[5] at text-
File at  < console > :22
scala >  t. count
15/04/19 08:16:47 INFOFileInputFormat:Total input paths to process:1
15/04/19 08:16:47 INFO:Tachyon client ( version 0. 6. 3 ) is trying to connect master @ cluster04/
192. 168. 242. 135:19998
15/04/19 08 : 16 : 47  INFO: User registered at the master cluster04/192. 168. 242. 135 : 19998 got
UserId 22
15/04/19 08:16:47 INFO:Trying to get local worker host:cluster04
15/04/19 08:16:47 INFO:Connecting local worker @  cluster04/192. 168. 242. 135:29998
15/04/19 08:16:47INFO:Folder /mnt/ramdisk/tachyonworker/users/22 was created!
15/04/19 08:16:47INFO:/mnt/ramdisk/tachyonworker/users/22/31138512896 was created!
res6:Long = 41
```

首先从本地加载文件 log4j. properties，构建 RDD 实例 t。

执行导入语句 import org. apache. spark. storage. StorageLevel 后，可以使用存储级别的定义。然后使用 StorageLevel. OFF_HEAP 存储级别，将 t 缓存到 Tachyon 中，最后通过 t. count 方法，触发缓存。

查看 Tachyon 的 Web Interface 界面（http://cluster04:19999）的 In Memory Files 页面，如图 5. 26 所示。

可以看到，RDD 已经成功缓存到 Tachyon 文件系统中。其中，缓存路径/spark/部分，是配置参数中设置的 park. tachyonStore. baseDir 值，可以是以逗号分割的多个目录路径。当该 spark - shell 结束时，RDD 会被自动清理。

7. Master 容错下的多 Master 访问

当 Tachyon 使用 ZooKeeper 处理 Master 容错时，可以通过任意一个 Master 来访问 Tachyon 文件系统。

首先需要添加 ZooKeeper 相关的配置属性，在 conf/spark - env. sh 中添加：

```
Export Spark_JAVA_OPTS = "
    - Dtachyon. zookeeper. address = cluster05:2181
    - Dtachyon. usezookeeper = true
    $ Spark_JAVA_OPTS"
```

File Path	Size	Block Size	Pin	Creation Time	Modification Time
/my_tachyon-env.sh/_SUCCESS	0.00 B	512.00 MB	NO	04-19-2015 05:50:00:411	04-19-2015 05:50:00:456
/my_tachyon-env.sh/part-00000	3525.00 B	512.00 MB	NO	04-19-2015 05:49:59:766	04-19-2015 05:50:00:255
/save1/_SUCCESS	0.00 B	512.00 MB	NO	04-19-2015 08:15:41:463	04-19-2015 08:15:41:510
/save1/part-00000	3525.00 B	512.00 MB	NO	04-19-2015 08:15:41:081	04-19-2015 08:15:41:401
/spark/spark-24fcda62-558f-4f88-a49d-7ece0c2533f7//spark-tachyon-20150419081647-e5a5/19/rdd_5_0	2057.00 B	512.00 MB	NO	04-19-2015 08:16:47:686	04-19-2015 08:16:47:703
/tachyon-env.sh	3525.00 B	512.00 MB	NO	04-19-2015 05:24:17:689	04-19-2015 05:24:21:672

View Settings

图 5.26 StorageLevel. OFF_HEAP 存储级别缓存后的界面

这种设置方式在运行时会出现以下警告信息：

```
Spark_JAVA_OPTS was detected (set to'
    - Dtachyon. zookeeper. address = cluster05: 2181
    - Dtachyon. usezookeeper = true

' ).
This is deprecated inSpark 1.0 +.
Please instead use:
    - ./spark - submit with conf/spark - defaults. conf to set defaults for an application
    - ./spark - submit with  -- driver - java - options to set  - X options for a driver
    - spark. executor. extraJavaOptions to set  - X options for executors
    - sPARK_DAEMON_JAVA_OPTS to set java options for standalone daemons (master or worker)
15/04/21 00:23:07 WARNSparkConf:Setting' spark. executor. extraJavaOptions' to'
    - Dtachyon. zookeeper. address = cluster05: 2181
    - Dtachyon. usezookeeper = true
'  as a work - around.
15/04/21 00:23:07 WARNSparkConf:Setting' spark. driver. extraJavaOptions' to'
    - Dtachyon. zookeeper. address = cluster05: 2181
    - Dtachyon. usezookeeper = true
```

警告信息可以忽略，如果将属性配置到 conf/spark - defaults. conf 的话，在 Tachyon 中是不起作用的。

另外在 conf/core - site. xml 文件中添加以下信息：

```
< property >
< name > fs. tachyon - ft. impl </name >
```

```
<value>tachyon.hadoop.TFS</value>
</property>
```

注意：在重新编译的 Spark 1.3.0 版本（基于 Hadoop 2.x 版本）中，没有设置也可以成功访问 Tachyon。

启动 spark-shell，用 Active Master 指定路径来获取文件内容，并保存到 Standby Master 指定的路径上。

```
scala>    val s = sc.textFile("tachyon-ft://cluster04:19998/tachyon-env.sh")
15/04/21 01:16:25 INFOSparkContext:Created broadcast 0 from textFile at <console>:21
s:org.apache.spark.rdd.RDD[String] = tachyon-ft://cluster04:19998/tachyon-env.sh MapPartitionsRDD[1] at textFile at <console>:21

scala>    s.count
15/04/21 01:16:26INFO:initialize(tachyon-ft://cluster04:19998/tachyon-env.sh,Configuration:core-default.xml,core-site.xml,mapred-default.xml,mapred-site.xml,yarn-default.xml,yarn-site.xml,hdfs-default.xml,hdfs-site.xml). Connecting to Tachyon:tachyon-ft://cluster04:19998/tachyon-env.sh
15/04/21 01:16:26WARN:tachyon.home is not set. Using /mnt/tachyon_default_home as the default value.
15/04/21 01:16:26 INFO CuratorFrameworkImpl:Starting
15/04/21 01:16:26 INFO ZooKeeper:Client environment:zookeeper.version=3.4.5-1392090,built on 09/30/2012 17:52 GMT
15/04/21 01:16:26 INFO ZooKeeper:Client environment:host.name=cluster04
15/04/21 01:16:26 INFO ZooKeeper:Client environment:java.version=1.7.0_71
15/04/21 01:16:26 INFO ZooKeeper:Client environment:java.vendor=Oracle Corporation
15/04/21 01:16:26 INFO ZooKeeper:Client environment:java.home=/lib/jdk1.7.0_71/jre
15/04/21 01:16:26 INFO ZooKeeper:Client environment:java.class.path=:/home/harli/cluster_13/spark/conf:/home/harli/cluster_13/spark/lib/spark-assembly-1.3.0-hadoop2.6.0.jar:/home/harli/cluster_13/spark/lib/datanucleus-rdbms-3.2.9.jar:/home/harli/cluster_13/spark/lib/datanucleus-core-3.2.10.jar:/home/harli/cluster_13/spark/lib/datanucleus-api-jdo-3.2.6.jar:/home/harli/cluster/hadoop/etc/hadoop
15/04/21 01:16:26 INFO ZooKeeper:Client environment:java.library.path=/usr/java/packages/lib/amd64:/usr/lib64:/lib64:/lib:/usr/lib
15/04/21 01:16:26 INFO ZooKeeper:Client environment:java.io.tmpdir=/tmp
15/04/21 01:16:26 INFO ZooKeeper:Client environment:java.compiler=<NA>
15/04/21 01:16:26 INFO ZooKeeper:Client environment:os.name=Linux
15/04/21 01:16:26 INFO ZooKeeper:Client environment:os.arch=amd64
15/04/21 01:16:26 INFO ZooKeeper:Client environment:os.version=2.6.32-431.el6.x86_64
15/04/21 01:16:26 INFO ZooKeeper:Client environment:user.name=harli
15/04/21 01:16:26 INFO ZooKeeper:Client environment:user.home=/home/harli
15/04/21 01:16:26 INFO ZooKeeper:Client environment:user.dir=/home/harli/cluster_13/spark
15/04/21 01:16:26 INFO ZooKeeper:Initiating client connection,connectString=cluster05:2181sessionTimeout=60000 watcher=org.apache.curator.ConnectionState@3eaf5470
15/04/21 01:16:26 INFOClientCnxn:Opening socket connection to server cluster05/192.168.242.136:
```



2181. Will not attempt to authenticate using SASL (unknown error)
15/04/21 01:16:26 INFOClientCnxn:Socket connection established to cluster05/192.168.242.136:2181,initiating session
15/04/21 01:16:26 INFOClientCnxn: Session establishment complete on server cluster05/192.168.242.136:2181,sessionid = 0x14cdab361fe000f,negotiated timeout = 60000
15/04/21 01:16:26 INFO ConnectionStateManager:State change:CONNECTED
15/04/21 01:16:26 WARN ConnectionStateManager:There are no ConnectionStateListeners registered.
15/04/21 01:16:27 INFO:Master addresses:[cluster04:19998,cluster06:19998]
15/04/21 01:16:27 INFO:Tachyon client (version 0.6.3) is trying to connect master @ cluster04/192.168.242.135:19998
15/04/21 01:16:27 INFO: User registered at the master cluster04/192.168.242.135:19998 got UserId 11
15/04/21 01:16:27 INFO:tachyon - ft://cluster04:19998 tachyon - ft://cluster04:19998hdfs://cluster04:9000
15/04/21 01:16:27 INFO:getFileStatus(tachyon - ft://cluster04:19998/tachyon - env.sh):HDFS Path:hdfs://cluster04:9000/tachyon - env.sh TPath:tachyon - ft://cluster04:19998/tachyon - env.sh
15/04/21 01:16:27 INFOFileInputFormat:Total input paths to process:1
15/04/21 01:16:27 INFOSparkContext:Starting job:count at <console>:24
15/04/21 01:16:27 INFODAGScheduler:Got job 0 (count at <console>:24) with 1 output partitions (allowLocal = false)
15/04/21 01:16:27 INFODAGScheduler:Final stage:Stage 0(count at <console>:24)
15/04/21 01:16:27 INFODAGScheduler:Parents of final stage:List()
15/04/21 01:16:27 INFODAGScheduler:Missing parents:List()
15/04/21 01:16:27 INFODAGScheduler:Submitting Stage 0 (tachyon - ft://cluster04:19998/tachyon - env.sh MapPartitionsRDD[1] at textFile at <console>:21),which has no missing parents
15/04/21 01:16:27 INFOSparkContext:Created broadcast 1 from broadcast at DAGScheduler.scala:839
15/04/21 01:16:27 INFODAGScheduler:Submitting 1 missing tasks from Stage 0 (tachyon - ft://cluster04:19998/tachyon - env.sh MapPartitionsRDD[1] at textFile at <console>:21)
15/04/21 01:16:27 INFOTaskSchedulerImpl:Adding task set 0.0 with 1 tasks
15/04/21 01:16:27 INFOTaskSetManager:Starting task 0.0 in stage 0.0 (TID 0,localhost,ANY,1307 bytes)
15/04/21 01:16:27 INFO Executor:Running task 0.0 in stage 0.0 (TID 0)
15/04/21 01:16:27 INFOHadoopRDD:Input split:tachyon - ft://cluster04:19998/tachyon - env.sh:0 + 3647
15/04/21 01:16:27 INFO deprecation:mapred.tip.id is deprecated. Instead,use mapreduce.task.id
15/04/21 01:16:27 INFO deprecation: mapred.task.id is deprecated. Instead, use mapreduce.task.attempt.id
15/04/21 01:16:27 INFO deprecation: mapred.task.is.map is deprecated. Instead, use mapreduce.task.ismap
15/04/21 01:16:27 INFO deprecation: mapred.task.partition is deprecated. Instead, use mapreduce.task.partition
15/04/21 01:16:27 INFO deprecation:mapred.job.id is deprecated. Instead,use mapreduce.job.id
15/04/21 01:16:27 INFO:open(tachyon - ft://cluster04:19998/tachyon - env.sh,65536)
15/04/21 01:16:27 INFO:Trying to get local worker host:cluster04
15/04/21 01:16:27 INFO:Connecting local worker @ cluster04/192.168.242.135:29998

```
15/04/21 01:16:27 INFO Executor:Finished task 0.0 in stage 0.0 (TID 0). 1830 bytes result sent to driver
15/04/21 01:16:27 INFODAGScheduler:Stage 0 (count at <console>:24) finished in 0.138 s
15/04/21 01:16:27 INFOTaskSetManager:Finished task 0.0 in stage 0.0 (TID 0) in 126 ms on localhost (1/1)
15/04/21 01:16:27 INFOTaskSchedulerImpl:Removed TaskSet 0.0, whose tasks have all completed, from pool
15/04/21 01:16:27 INFODAGScheduler:Job 0 finished:count at <console>:24, took 0.261197 s
res0:Long = 90

scala>  s.saveAsTextFile("tachyon-ft://cluster06:19998/save1")
15/04/21 01:16:46INFO:initialize(tachyon-ft://cluster06:19998/save1, Configuration:core-default.xml, core-site.xml, mapred-default.xml, mapred-site.xml, yarn-default.xml, yarn-site.xml, hdfs-default.xml, hdfs-site.xml). Connecting to Tachyon:tachyon-ft://cluster06:19998/save1
15/04/21 01:16:46 INFO:Master addresses:[cluster04:19998, cluster06:19998]
15/04/21 01:16:46 INFO:Tachyon client (version 0.6.3) is trying to connect master @ cluster04/192.168.242.135:19998
15/04/21 01:16:46 INFO:User registered at the master cluster04/192.168.242.135:19998 got UserId 12
15/04/21 01:16:46 INFO:tachyon-ft://cluster06:19998 tachyon-ft://cluster06:19998hdfs://cluster04:9000
15/04/21 01:16:46 INFO:getWorkingDirectory:/
15/04/21 01:16:46 INFO:getWorkingDirectory:/
15/04/21 01:16:46 INFO:getFileStatus(tachyon-ft://cluster06:19998/save1):HDFS Path:hdfs://cluster04:9000/save1 TPath:tachyon-ft://cluster06:19998/save1
15/04/21 01:16:47 INFOContextCleaner:Cleaned broadcast 1
15/04/21 01:16:47 INFO:File does not exist:tachyon-ft://cluster06:19998/save1
15/04/21 01:16:47 INFO:getWorkingDirectory:/
15/04/21 01:16:47 INFO:mkdirs(tachyon-ft://cluster06:19998/save1/_temporary/0, rwxrwxrwx)
15/04/21 01:16:47 INFOSparkContext:Starting job:saveAsTextFile at <console>:24
15/04/21 01:16:47 INFODAGScheduler:Got job 1 (saveAsTextFile at <console>:24) with 1 output partitions (allowLocal = false)
15/04/21 01:16:47 INFODAGScheduler:Final stage:Stage 1(saveAsTextFile at <console>:24)
15/04/21 01:16:47 INFODAGScheduler:Parents of final stage:List()
15/04/21 01:16:47 INFODAGScheduler:Missing parents:List()
15/04/21 01:16:47 INFODAGScheduler:Submitting Stage 1 (MapPartitionsRDD[2] at saveAsTextFile at <console>:24), which has no missing parents
15/04/21 01:16:47 INFOSparkContext:Created broadcast 2 from broadcast at DAGScheduler.scala:839
15/04/21 01:16:47 INFODAGScheduler:Submitting 1 missing tasks from Stage 1 (MapPartitionsRDD[2] at saveAsTextFile at <console>:24)
15/04/21 01:16:47 INFOTaskSchedulerImpl:Adding task set 1.0 with 1 tasks
15/04/21 01:16:47 INFOTaskSetManager:Starting task 0.0 in stage 1.0 (TID 1, localhost, ANY, 1307 bytes)
15/04/21 01:16:47 INFO Executor:Running task 0.0 in stage 1.0 (TID 1)
15/04/21 01:16:47 INFOHadoopRDD:Input split:tachyon-ft://cluster04:19998/tachyon-env.sh:0
```



+3647
15/04/21 01:16:47 INFO:open(tachyon-ft://cluster04:19998/tachyon-env.sh,65536)
15/04/21 01:16:47 INFO:getWorkingDirectory:/
15/04/21 01:16:47 INFO:create(tachyon-ft://cluster06:19998/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1/part-00000,rw-r--r--,true,65536,1,536870912,org.apache.hadoop.mapred.Reporter $ 1@3bfc246d)
15/04/21 01:16:47 INFO:Trying to get local worker host:cluster04
15/04/21 01:16:47 INFO:Connecting local worker @ cluster04/192.168.242.135:29998
15/04/21 01:16:47INFO:Folder /mnt/ramdisk/tachyonworker/users/12 was created!
15/04/21 01:16:47INFO:/mnt/ramdisk/tachyonworker/users/12/23622320128 was created!
15/04/21 01:16:48 INFO:getFileStatus(tachyon-ft://cluster06:19998/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1 TPath:tachyon-ft://cluster06:19998/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1
15/04/21 01:16:48 INFO:getFileStatus(tachyon-ft://cluster06:19998/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1 TPath:tachyon-ft://cluster06:19998/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1
15/04/21 01:16:48 INFO:getFileStatus(tachyon-ft://cluster06:19998/save1/_temporary/0/task_201504210116_0001_m_000000):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0/task_201504210116_0001_m_000000 TPath:tachyon-ft://cluster06:19998/save1/_temporary/0/task_201504210116_0001_m_000000
15/04/21 01:16:48 INFO:File does not exist:tachyon-ft://cluster06:19998/save1/_temporary/0/task_201504210116_0001_m_000000
15/04/21 01:16:48 INFO:rename(tachyon-ft://cluster06:19998/save1/_temporary/0/_temporary/attempt_201504210116_0001_m_000000_1,tachyon-ft://cluster06:19998/save1/_temporary/0/task_201504210116_0001_m_000000)
15/04/21 01:16:48 INFOFileOutputCommitter:Saved output of task' attempt_201504210116_0001_m_000000_1' to tachyon-ft://cluster06:19998/save1/_temporary/0/task_201504210116_0001_m_000000
15/04/21 01:16:48 INFOSparkHadoopWriter:attempt_201504210116_0001_m_000000_1:Committed
15/04/21 01:16:48 INFO Executor:Finished task 0.0 in stage 1.0 (TID1). 2085 bytes result sent to driver
15/04/21 01:16:48 INFODAGScheduler:Stage 1 (saveAsTextFile at <console>:24) finished in 0.490 s
15/04/21 01:16:48 INFODAGScheduler:Job 1 finished:saveAsTextFile at <console>:24,took 0.612670 s
15/04/21 01:16:48 INFO:listStatus(tachyon-ft://cluster06:19998/save1/_temporary/0):HDFS Path:hdfs://cluster04:9000/save1/_temporary/0
15/04/21 01:16:48 INFOTaskSetManager:Finished task 0.0 in stage 1.0 (TID 1) in 499 ms on localhost (1/1)
15/04/21 01:16:48 INFOTaskSchedulerImpl:Removed TaskSet 1.0,whose tasks have all completed, from pool
15/04/21 01:16:48 INFO:getFileStatus(tachyon-ft://cluster06:19998/save1):HDFS Path:hdfs://cluster04:9000/save1 TPath:tachyon-ft://cluster06:19998/save1
15/04/21 01:16:48 INFO:getFileStatus(tachyon-ft://cluster06:19998/save1):HDFS Path:hdfs://

```
cluster04:9000/save1 TPath:tachyon - ft://cluster06:19998/save1
15/04/21 01:16:48 INFO: listStatus(tachyon - ft://cluster06:19998/save1/_temporary/0/task_201504210116_0001_m_000000): HDFS Path: hdfs://cluster04:9000/save1/_temporary/0/task_201504210116_0001_m_000000
15/04/21 01:16:48 INFO:getFileStatus(tachyon - ft://cluster06:19998/save1/part - 00000):HDFS Path:hdfs://cluster04:9000/save1/part - 00000 TPath: tachyon - ft://cluster06:19998/save1/part - 00000
15/04/21 01:16:48 INFO:File does not exist:tachyon - ft://cluster06:19998/save1/part - 00000
15/04/21 01:16:48 INFO: rename(tachyon - ft://cluster06:19998/save1/_temporary/0/task_201504210116_0001_m_000000/part - 00000,tachyon - ft://cluster06:19998/save1/part - 00000)
15/04/21 01:16:48 INFO:delete(tachyon - ft://cluster06:19998/save1/_temporary,true)
15/04/21 01:16:48 INFO:create(tachyon - ft://cluster06:19998/save1/_SUCCESS,rw - r - - r - - ,true,65536,1,536870912,null)
```

可以看到，这里使用了 tachyon - ft://cluster04:19998 和 tachyon - ft://cluster06:19998（其中作为 Scheme 信息的 tachyon - ft 中，ft 表示 fault tolerant），访问了 Active 和 Standby 的 Master 指定路径。

日志中的“Master addresses:[cluster04:19998,cluster06:19998]”，是根据 ZooKeeper 相关属性获取的当前 Master 列表信息。之后会尝试连接到 Active Master，然后进行具体操作。

保存结果可以查看对应的 Web Interface 界面，如图 5.27 所示。

File Path	Size	Block Size	Pin	Creation Time	Modification Time
/Y/_SUCCESS	0.00 B	512.00 MB	NO	04-21-2015 00:13:24:310	04-21-2015 00:13:24:347
/Y/part-00000	3647.00 B	512.00 MB	NO	04-21-2015 00:13:23:806	04-21-2015 00:13:24:217
/save1/_SUCCESS	0.00 B	512.00 MB	NO	04-21-2015 01:16:48:203	04-21-2015 01:16:48:235
/save1/part-00000	3647.00 B	512.00 MB	NO	04-21-2015 01:16:47:835	04-21-2015 01:16:48:117
/tachyon-env.sh	3647.00 B	512.00 MB	NO	04-21-2015 00:02:12:578	04-21-2015 00:02:15:720
/tachyon-save/_SUCCESS	0.00 B	512.00 MB	NO	04-21-2015 00:25:31:144	04-21-2015 00:25:31:169
/tachyon-save/part-00000	3647.00 B	512.00 MB	NO	04-21-2015 00:25:30:788	04-21-2015 00:25:31:039

图 5.27　Tachyon 文件系统的 tachyon - ft 方式访问界面

注意，如果使用的是没有重新编译过的 Spark 1.3.0，比如从官网直接下载时，在 Tachyon 上执行 Spark 时，会出现以下错误。

1. 错误一：

```
scala > s.count()
```

```
15/04/19 05:29:51INFO:initialize(tachyon://cluster04:19998/tachyon-env.sh,Configuration:core-default.xml,core-site.xml,mapred-default.xml,mapred-site.xml,yarn-default.xml,yarn-site.xml,hdfs-default.xml,hdfs-site.xml). Connecting to Tachyon:tachyon://cluster04:19998/tachyon-env.sh
15/04/19 05:29:52 INFO:Trying to connect master @ cluster04/192.168.242.135:19998
15/04/19 05:29:52 INFO:User registered at the master cluster04/192.168.242.135:19998 got UserId 6
15/04/19 05:29:52 INFO:Trying to get local worker host:cluster04
15/04/19 05:29:52 INFO:Connecting local worker @ cluster04/192.168.242.135:29998
15/04/19 05:29:52 ERROR:Invalid method name': getDataFolder'
java.io.IOException:Invalid method name': user_getUnderfsAddress'
    at tachyon.client.TachyonFS.getUnderfsAddress(TachyonFS.java:1228)
    at tachyon.hadoop.TFS.initialize(TFS.java:289)
```

必须在 spark-env.sh 文件中配置 Spark_CLASSPATH，具体值参看前面内容。

2. 错误二：

```
scala> t.count
15/04/19 06:02:17 INFOFileInputFormat:Total input paths to process:1
15/04/19 06:02:17 INFO deprecation:mapred.tip.id is deprecated. Instead,use mapreduce.task.id
15/04/19 06:02:17 INFO deprecation:mapred.task.id is deprecated. Instead,use mapreduce.task.attempt.id
15/04/19 06:02:17 INFO deprecation:mapred.task.is.map is deprecated. Instead,use mapreduce.task.ismap
15/04/19 06:02:17 INFO deprecation:mapred.task.partition is deprecated. Instead,use mapreduce.task.partition
15/04/19 06:02:17 INFO deprecation:mapred.job.id is deprecated. Instead,use mapreduce.job.id
15/04/19 06:02:17WARN:tachyon.home is not set. Using /mnt/tachyon_default_home as the default value.
15/04/19 06:02:17 WARNBlockManager:Putting block rdd_3_0 failed
15/04/19 06:02:17 ERROR Executor:Exception in task 0.0 in stage 0.0 (TID 0)
java.lang.NoSuchMethodError:tachyon.client.TachyonFS.exist(Ljava/lang/String;)Z
at org.apache.spark.storage.TachyonBlockManager$$anonfun$createTachyonDirs$2.apply(TachyonBlockManager.scala:117)
at org.apache.spark.storage.TachyonBlockManager$$anonfun$createTachyonDirs$2.apply(TachyonBlockManager.scala:106)
at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:244)
at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:244)
```

该错误是由于下载的 Spark 1.3.0 依赖的 Tachyon 版本为 1.5.0，和 Tachyon 1.6.0 版本不兼容，必须进行重编译后才能运行通过。

5.7.2 基于 Tachyon 运行 HADOOP MR 的案例与解析

进行 Tachyon 上运行 HadoopMapReduce 的案例实践时，已经部署了 Tachyon 集群，并且

配置了容错处理，即启动了 ZooKeeper 对 Master 进行管理，启动了 HDFS，同时将 HDFS 配置为 Tachyon 的底层文件系统。

使用 Hadoop 提供的 WordCount 案例进行测试：

```
[harli@ cluster04 hadoop] $ ./bin/hadoop jar ./share/hadoop/mapreduce/hadoop - mapreduce - examples - 2.6.0.jar wordcount  - libjars ../tachyon/client/target/tachyon - client - 0.6.3 - jar - with - dependencies.jar   tachyon://cluster04:19998/tachyon - env.sh hdfs://cluster04:9000/output
15/04/21 08:48:16INFO:initialize(tachyon://cluster04:19998/tachyon - env.sh,Configuration:core - default.xml,core - site.xml, mapred - default.xml, mapred - site.xml, yarn - default.xml, yarn - site.xml,hdfs - default.xml,hdfs - site.xml). Connecting to Tachyon:tachyon://cluster04:19998/tachyon - env.sh
15/04/21 08:48:16WARN:tachyon.home is not set. Using /mnt/tachyon_default_home as the default value.
15/04/21 08:48:16 INFO:Tachyon client (version 0.6.3) is trying to connect master @ cluster04/192.168.242.135:19998
15/04/21 08:48:17 INFO:User registered at the master cluster04/192.168.242.135:19998 got UserId 3
15/04/21 08:48:17 INFO: tachyon://cluster04:19998 tachyon://cluster04:19998hdfs://cluster04:9000
15/04/21 08:48:17 INFO:getWorkingDirectory:/
15/04/21 08:48:17 INFO client.RMProxy: Connecting to ResourceManager at cluster04/192.168.242.135:8032
15/04/21 08:48:21 INFO:getFileStatus(tachyon://cluster04:19998/tachyon - env.sh):HDFS Path:hdfs://cluster04:9000/tachyon - env.sh TPath:tachyon://cluster04:19998/tachyon - env.sh
15/04/21 08:48:21 INFO input.FileInputFormat:Total input paths to process:1
15/04/21 08:48:21 INFOmapreduce.JobSubmitter:number of splits:1
15/04/21 08:48:22 INFOmapreduce.JobSubmitter:Submitting tokens for job:job_1429631282514_0001
15/04/21 08:48:22 INFOimpl.YarnClientImpl:Submitted application application_1429631282514_0001
15/04/21 08:48:22 INFOmapreduce.Job:The url to track the job:http://cluster04:8088/proxy/application_1429631282514_0001/
15/04/21 08:48:22 INFOmapreduce.Job:Running job:job_1429631282514_0001
15/04/21 08:48:36 INFOmapreduce.Job:Job job_1429631282514_0001 running in uber mode:false
15/04/21 08:48:36 INFOmapreduce.Job:  map 0%  reduce 0%
15/04/21 08:48:45 INFOmapreduce.Job:  map 100%  reduce 0%
15/04/21 08:48:57 INFOmapreduce.Job:  map 100%  reduce 100%
15/04/21 08:48:57 INFOmapreduce.Job:Job job_1429631282514_0001 completed successfully
15/04/21 08:48:57 INFOmapreduce.Job:Counters:54
```

这里使用了 - libjars 选项，指定用到的 Tachyon 客户端访问的依赖包 tachyon - client - 0.6.3 - jar - with - dependencies.jar，Hadoop 会将该 jar 包分发到各个任务节点上后，并自动添加到任务的 CLASSPATH 环境变量中

查看 HDFS 文件系统的 Web Interface（http://cluster04:50070/explorer.html#/）界面，如图 5.28 所示。

cluster04:50070/explorer.html#/output

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Browse Directory

/output

Permission	Owner	Group	Size	Replication	Block Size	Name
-rw-r--r--	harli	supergroup	0 B	1	128 MB	_SUCCESS
-rw-r--r--	harli	supergroup	3.32 KB	1	128 MB	part-r-00000

图 5. 28　Tachyon 文件系统在 Hadoop MR 访问后的界面

由图 5. 28 可以看到输出已经成功。

附录　Spark 1.4版本新特性



目前 Spark 基本上以 3 个月为周期发布一个 Release 版本，这可以从 Spark 的官方网站上看到，截稿之前，最新的发布版本已经更新到 Spark 1.4。即，在 2015 年 7 月 11 日（美国时间），Spark 1.4 版本正式发布，新版本除了对 Spark Core、Spark SQL（DataFrame）、Spark Streaming、Spark ML/MLlib 等进行缺陷（BUG）修复与升级之外，还加入了期待已久的 SparkR 组件。



Spark1.4 版本的新特性可以参考官网上 ReleaseNote 内容，下面简单介绍备受瞩目的两个新特性：Spark 1.4 版本中的 SparkR 组件和 Spark UI 的数据可视化功能。

1. SparkR 简介

和 Spark 一样，SparkR 也是 AMPLab 下的子项目，是整合 R 的易用性和 Spark 的扩展性的一个探索。SparkR 开发者的预览版最早在 2014 年 1 月开放源代码（http://amplab-extras.github.io/SparkR-pkg/）。随后的一年多时间，SparkR 在 AMPLab 得到了飞速发展，而在许多贡献者的努力下，SparkR 在性能和可用性上得到了显著提升。最新发布的 Spark 1.4 版本中已经将 SparkR 合并进来，作为 Alpha 组件发布。

SparkR 组件的安装可以参考 SparkR-pkg 官网上的描述。

在 Spark 1.4 中，SparkR 的核心组件是 SparkR DataFrames——在 Spark 上实现的一个分布式的抽象数据 DataFrame。DataFrame 是 R 中处理数据的基本数据结构，现在这个概念在很多框架上使用，SparkR 引入了 DataFrame 这一概念后，就更加扩展了可集成的数据源，尤其是面向数据科学家，其易用性更是大大提高了。

DataFrame 的使用案例可以参考 Spark 官网上的 SparkR 编程指南部分。

关于 SparkR 的更详细的信息，可以参考 Databricks 公司的网站（http://databricks.com/blog/2015/06/09/announcing-sparkr-r-on-spark.html）。

2. Spark UI 的数据可视化

最新发布的 Spark 1.4 版本，为 Spark UI 注入了新特性，即数据可视化。数据可视化不仅仅提高了开发者调试和性能调优的便利性，也极大地方便了开发者对于 Spark 内部 DAG 调度机制、RDD 缓存等内容的理解。

数据可视化包含以下三个部分：

1）Spark events 时间轴视图：Spark 最新的 1.4 版本中，Spark UI 将 Spark 的 events 基于一个时间轴进行显示，让用户可以一眼区别相对和交叉顺序。

2）Execution DAG：在 Spark 最新的 1.4 版本中，这个可视化聚焦于 DAG 执行的每个作业。在 Spark 中，job 与被组织在 DAG 中的一组 RDD 依赖性密切相关，最新的 Spark UI 会详细显示出 DAG 调度机制的内部细节，包括 stage 的划分，RDD 间的依赖关系，甚至还包含了 RDD 是否被缓存等信息。

3）Spark Streaming 统计数字可视化：Streaming 在这个版本中增加了新的 UI，各种详细信息尽收眼底。另外此版本也支持了 0.8.2.x 的 Kafka 版本。

其中，1）和 2）两部分的内容可以参考 Databricks 公司的博客（https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html）。

3）部分的内容可以参考 Databricks 公司的博客（https://www.databricks.com/blog/2015/07/08/new-visualizations-for-understanding-spark-streaming-applications.html）。

3. 2015 年大数据领域的两大峰会

Hadoop Summit 和 Spark Summit 可以说是大数据领域的两大盛会。随着大数据的社区规模不断扩大。今年这两大峰会的参会人数都创了新高。

Hadoop Summit 2015 的参会人数是 4000 人，同比增长 30%（2014：3100 人，2013：2600 人，2012：2100 人，2011：1600 人，2010：1200 人）。

Spark Summit 2015 的参会人数是 2000 人，同比增长 300%（2014：500 人）。

从参会人数我们就可以看出 Hadoop 和 Spark 这两大框架在大数据领域的重要性，以及目前大数据的热门程度了。虽然在总的参会人数上，Spark 峰会的人数远远不如 Hadoop 峰会，但就其同比增长的速度，完全体现了 Spark 计算框架的未来发展势头。下图是 Databricks 公司分享的 Spark 应用情况。

目前，Spark 最大的集群来自腾讯，集群包含了 8000 个节点，而单个 Job 最大分别是来自阿里巴巴和 Databricks，数据量为 1PB。

作者沟通交流方式：

微信公众号：DT_Spark 微信号：18610086859

新浪微博：@ilovepains

书籍QQ交流群：

DT大数据梦工厂①：462923555

DT大数据梦工厂②：437123764

DT大数据梦工厂③：418110145

- 本书是面向Spark开发者的一本实用参考书，书中结合实例系统地介绍了Spark的开发与使用。
- 本书共5章，第1章为Spark简介；第2章为Spark RDD实践案例与解析；第3章为Spark SQL实践案例与解析；第4章为Spark Streaming实践案例与解析；第5章为Tachyon实战案例与解析。在全书最后的附录部分介绍了Spark1.4版本的新特性。
- 本书适合刚接触Spark或对Spark分布式计算的开发不熟悉的初学者学习。对于熟悉函数式开发或面向对象开发，并有一定经验的开发者，本书也可以作为参考。

Spark 大数据 实例开发教程



地址：北京市百万庄大街22号

邮政编码：100037

电话服务

服务咨询热线：010-88361066

读者购书热线：010-68326294

010-88379203

网络服务

机工官网：www.cmpbook.com

机工官博：weibo.com/cmp1952

金书网：www.golden-book.com

教育服务网：www.cmpedu.com

封面无防伪标均为盗版





机械工业出版社微信公众号/机械工业出版社计算机分社微信服务号

上架指导 计算机/大数据

ISBN 978-7-111-51909-6

策划编辑◎王斌 / 封面设计◎史淑娴



定价：59.00元